THE EXPERT'S VOICE® IN OPEN SOURCE

The Definitive Guide to SQLite

Take control of this compact but powerful tool to embed sophisticated SQL databases within your applications!

Michael Owens

Foreword by D. Richard Hipp, creator and lead developer of SQLite

Apress®

目錄

SQLite权威指南

The Definitive Guide to SQLite

（内容摘要）

Michael Owens

本书的示例代码可到http://www.apress.com下载。

推荐者的话

最近对SQLite很感兴趣，认真学习有一个多月了。

学习时基本找不到既好又系统的中文文章，也买不到好的中文书籍，看来SQLite在国内还是不够流行，这么好的东西，可惜了。

以我中等偏下的眼界，《The Definitive Guide to SQLite》是我所见到的最好的“SQLite入门+大全”了，可惜也是英文的。实在找不到别的，也只好看它了，尽管我英语很不好。

由于英语很不好，又因为是打字员出身的干部，所以多年来养成了一个更不好的毛病，就是在不得不看英文资料时总喜欢一边看一边翻，主要是怕下次再看时还是看不懂。看《The Definitive Guide to SQLite》时这个毛病也没改，当然了，看的时候就是挑着看的，翻的也只是书中的一小部分了。

一般情况下看完也就看完了，很少有“下次再看”的机会，这次例外。由于越学越觉着SQLite好玩，就想向身边的人也介绍一下，就“再看”了。越看越羞愧，本来英语就差，还随看随翻，结果可想而知。但由于没什么动力，也就无意再重新润色了，就这样吧，反正也没什么人看，估计遗害不会太广。

SQLite是没有版权的，但这本书却是受版权保护的，也不知我这样做是否合法。发到网上也只是想做一个好事，响应SQLite的共享精神。估计不会有人来告我吧，反正我没钱。另外，我也只翻译了书中很小的一部分，也许根本算不上翻译（不能乱抬高自己），就算是对SQLite和《The Definitive Guide to SQLite》一书的一个推荐吧，同样对SQLite感兴趣但又看不懂我的中文的兄弟，强烈建议看原文。感谢Richard Hipp编出这么好的程序，感谢Michael Owens写出这么好的书。

“空转”只是我的网名之一，网上网下知之者甚少，也就是一起骑车的几个人知道吧。如果本文对您能有一点点帮助，也算是我对SQLite做了一点贡献吧。本文中带有“空注”的内容是我个人所做的简单说明和忏悔，与原作者无关（以我的翻译水平，估计全文跟原作者都没什么关）。

接触SQLite时间不长，所以本文难免会有很多错误，不是故意误导大家，是真的水平低。如果有兄弟想对我提出指导，我的邮箱是：njgaoyi@yahoo.com.cn。如果我没有回信，不是因为不想回，是因为我很少上网，在此先行谢过。

分析源程序时，发现每个SQLite源文件的头部都有这样一段话：

The author disclaims copyright to this source code. In place of a legal notice, here is a blessing:

May you do good and not evil.

May you find forgiveness for yourself and forgive others.

May you share freely, never taking more than you give.

这几句话我很喜欢，翻译不好，就拿原文出来吧，与大家共勉。

空转

Ver 1.00: 2009-11-07 于南京

（如果以后有时间、兴趣，就把翻译过的内容好好修改一下，或者再多翻一些。但愿还有以后的版本）

前言

2000年春天，当我刚开始编写SQLite时，根本没想到它会在编程社区受到如此强烈的认可。今天，有成百万的SQLite拷贝在默默地运行，在计算机中，或在不同公司生产的各种各样的小设备中。你可能已经在无意识的情况下使用过SQLite，在你的手机、MP3或机顶盒里可能就有SQLite。在你的计算机里也可能至少会有一个SQLite的拷贝，它可能来自Apple的Mac OS X，或者在大多数的Linux版本中，或者在Windows中安装某个第三方软件时。很多Web网站的后台都使用SQLite，这要感谢它已经被包含为PHP5语言的一部分。SQLite也被用于很多航空电子设备、建模和仿眞程序、工业控制、智能卡、决策支持包、医药信息系统等。因为没有SQLite使用的全面报告，所以，肯定还有很多我不知道的SQLite部署。

SQLite的普及很大程度上应该归功于Michael Owens。Mike在The Linux Journal (June 2003)和The C/C++ Users Journal (March 2004)上的文章吸引了无数程序员。每篇文章发表后，SQLite网站的访问量都会显著上升。通过这本书你可以看到Mike的才华和他所做的大量工作，相信你不会失望。本书包含了关于SQLite所需要了解的所有内容，你应该一直把它放在伸手可及的地方。

SQLite是自由软件。尽管我是它的架构师和代码的主要编写者，但SQLite并不是我的程序。SQLite不属于任何人，也不在版权的保护范围之内。所有曾经为SQLite项目贡献过代码的人都签署过一个宣誓书将他们的贡献发布到公共域，我把这些宣誓书的原件保存在办公室的保险箱里。我还尽力保证在SQLite中不使用专利算法，这些预防措施意味着你可以以任何形式使用SQLite，而不需要付版税、许可证费用或受到其它任何限制。

SQLite仍然在发展。但我和其他开发者都坚守它的核心价值。我们将保持代码的小规模——核心库不会超过250KB。我们将保持公共API和文件格式的向上兼容性。我们将继续保证SQLite是充分测试的和无bug的。我们希望你总是能够将新版本的SQLite放到你老的程序中，既得到它新的特性和优化，又不需要或仅需要很少的代码改动，且不需要做进一步的调试。2004年，我们将SQLite从版本2升级到版本3时确实没能保持向上兼容性，但从那以后，我们已经能够达到上述所有目标并准备在将来继续这样做。没有SQLite版本4的计划。

眞诚希望你觉着SQLite是有用的，我代表SQLite的所有贡献者保证，使用SQLite你会：做出美好的产品，你的产品将会是快速、稳定和易用的。寻求宽恕并宽恕他人。因为你已经免费地得到了SQLite，也请你免费地给予他人一些东西作为回报。做一回志愿者，贡献出其它的软件项目或找到其它途径来回报。

Richard Hipp

Charlotte, NC

April 11, 2006

第1章 SQLite介绍

SQLite是一个开源的、内嵌式的关系型数据库。它最初发布于2000年，在便携性、易用性、紧凑性、有效性和可靠性方面有突出的表现。

内嵌式数据库

SQLite是一个内嵌式的数据库。

数据库服务器就在你的程序中，其好处是不需要网络配置和管理。数据库的服务器和客户端运行在同一个进程中。这样可以减少网络访问的消耗，简化数据库管理，使你的程序部署起来更容易。所有需要你做的都已经和你的程序一起编译好了。

如图1-1所示。一个Perl脚本、一个标准C/C++程序和一个使用PHP编写的Apache进程都使用SQLite。Perl脚本导入DBI::SQLite模板，并通过它来访问C API。PHP采用与C相似的方式访问C API。总之，它们都需要访问C API。尽管它们每个进程中都有独立的数据库服务器，但它们可以操作相同的数据库文件。SQLite利用操作系统功能来完成数据的同步和加锁。

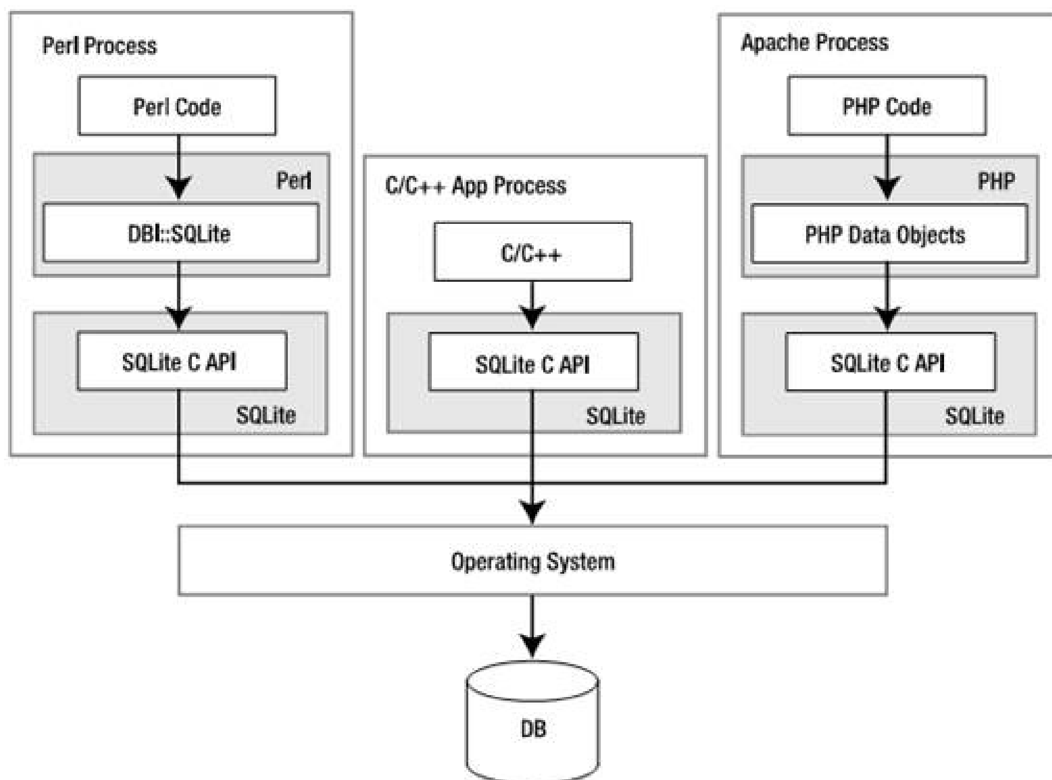


图1-1 内嵌的主进程中的SQLite

目前市场上有多种为内嵌应用所设计的关系型数据库产品，如Sybase SQL Anywhere、InterSystems Caché、Pervasive PSQL和微软的Jet Engine。有些厂家从他们的大型数据库产品翻新出内嵌式的变种，如IBM的DB2 Everyplace、Oracle的10g和微软的SQL Server

Desktop Engine。开源的数据库MySQL和Firebird都提供内嵌式的版本。在所有这些产品中，仅有两个是完全开放源代码的且不收许可证费用——Firebird和SQLite。在这两个当中，仅有一个是专门为内嵌式应用设计的——SQLite。

开发者的数据库

SQLite具有多方面的特性。它是一个数据库，一个程序库，一个命令行工具，也是一个学习关系型数据库的很好的工具。确实有很多途径可以使用它——内嵌环境、网站、操作系统服务、脚本语言和应用程序。对于程序员来说，SQLite就象一个数据传送带，提供了一种方便的将应用程序绑定的数据的方法。就象传送带一样，对SQLite的使用没有终点。

除了仅仅作为一个存储容器，SQLite还可以作为一个单纯的数据处理的工具。如果大小和复杂性合适，使用SQLite可以很容易地将应用程序所使用的数据结构转化为表，并保存在一个内在数据库中。用此方法，你可以操作互相关联的数据，可以完成很繁重的任务页不必写自己的算法来对数据结构操作和排序。如果你是一个程序员，想像一下在你的程序中自行完成下面SQL语句所代表的工作需要多少代码：

```
SELECT AVG(z-y) FROM table GROUP BY x
    HAVING x > MIN(z) OR x < MAX(y)
    ORDER BY y DESC LIMIT 10 OFFSET 3;
```

SQLite还是一个很好的学习程序设计的工具，通过它可以研究很多计算机科学的课题。分析器、分词器、虚拟机、Btree算法、高整缓存、程序体系结构，通过这些内容可以搞清楚很多计算机科学的经典概念。SQLite的模块化、小型化和简易性，使你可以很容易地专门研究其中的一个问题。

管理员的数据库

SQLite不仅是程序员的数据库，它对系统管理员也很有用。它很小、紧凑而精致，就像一些Unix的常用工具，如find、rsync或grep。SQLite提供了命令行工具供用户交互操作。

另外，对于关系型数据库的初学者来说，SQLite是一个学习各种关系相关概念的方便的学习工具。它可以很快很容易地安装在各类操作系统中，它的数据库文件可以自由共享页不需要任何转换。它具有关系型数据库的各种特色而又不令人生畏。它的程序和数据库文件仅用U盘就能传递。

SQLite 的历史

从某个角度来说，SQLite最初的构思是在一条军舰上进行的。SQLite的作者D. Richard Hipp当时正在为美国海军编制一种使用在导弹驱逐舰上的程序。那个程序最初是运行在Hewlett-Packard Unix (HPUX)上，后台使用Informix数据库。对那个程序来说，Informix有点儿太强大

了。一个有经验的数据库管理员(DBA)可能需要一整天来对它进行安装和升级，如果没经验，这个工作就可能永远也做不完了。

2000年一月，Hipp开始和一个同事讨论关于创建一个简单的内嵌式SQL数据库的想法，这个数据库将使用GNU DBM B-Tree library (gdbm)做后台，同时这个数据库将不需要安装和管理支持。后来，当有些空闲时间时，Hipp就开始实施这项工作，并在2000年的八月份发布了SQLite的1.0版。

按照原定计划，SQLite 1.0用gdbm来做存储管理。但后来，Hipp很快就换成了自己的B-tree，以支持事务和记录按主键的存储。随着最初的升级，SQLite在功能和用户数上都得到了稳步的发展。在2001年中期，很多项目——开源的或商业的——都开始使用SQLite。在那以后的几年中，开源社区的其他成员开始为他们喜欢的程序设计语言编写SQLite扩展。SQLite的ODBC接口可以为Perl、Python、Ruby、Java和其它主流的程序设计语言提供支持，这证明了SQLite有广阔的应用前景。

2004年，SQLite从版本2升级到版本3，这是一次大升级。主要目的是增加内置的对UTF-8、UTF-16及用户定义字符集的支持。While 3.0 was originally slated for release in summer 2005, America Online provided the necessary funding to see that it was completed by July 2004. 除国际化功能外，版本3的其它新特性包括：经过修补的C API，更紧凑的数据库文件格式(比原来节省25%的空间)，弱类型，大二进制对象(BLOB)的支持，64-bit的ROWID，autovacuum和改进了的并发控制。尽管增加了这一系列新特性，版本3的运行库仍然小于240K字节。Another improvement in version 3 was a good code cleanup—revisiting and rewriting, or otherwise throwing out extraneous stuff accumulated in the 2.x series.

SQLite持续增长并始终坚持其最初的设计目标：简单、弹性、紧凑、速度和彻底的易用。本书出版时，SQLite已经增加了CHECK约束，下面就要增加外键约束，再下面呢？

谁使用**SQLite**

当前，SQLite已经被多种软件和产品所使用。它被用在Apple的Mac OS X操作系统中，被用作其CoreData应用程序架构的一部分。它还应用于Safari的Web浏览器、Mail.app的电子邮件程序、RSS的管理、Apple的Aperture照片软件。

尽管SQLite很少做广告，但它还是被用在了多种消费类产品中。

体系结构

SQLite拥有一个精致的、模块化的体系结构，并引进了一些独特的方法进行关系型数据库的管理。它由被组织在3个子系统中的8个独立的模块组成，如图1-2所示。这个模型将查询过程划分为几个不连续的任务，就像在流水线上工作一样。在体系结构栈的顶部编译查询语句，在中部执行它，在底部处理操作系统的存储和接口。

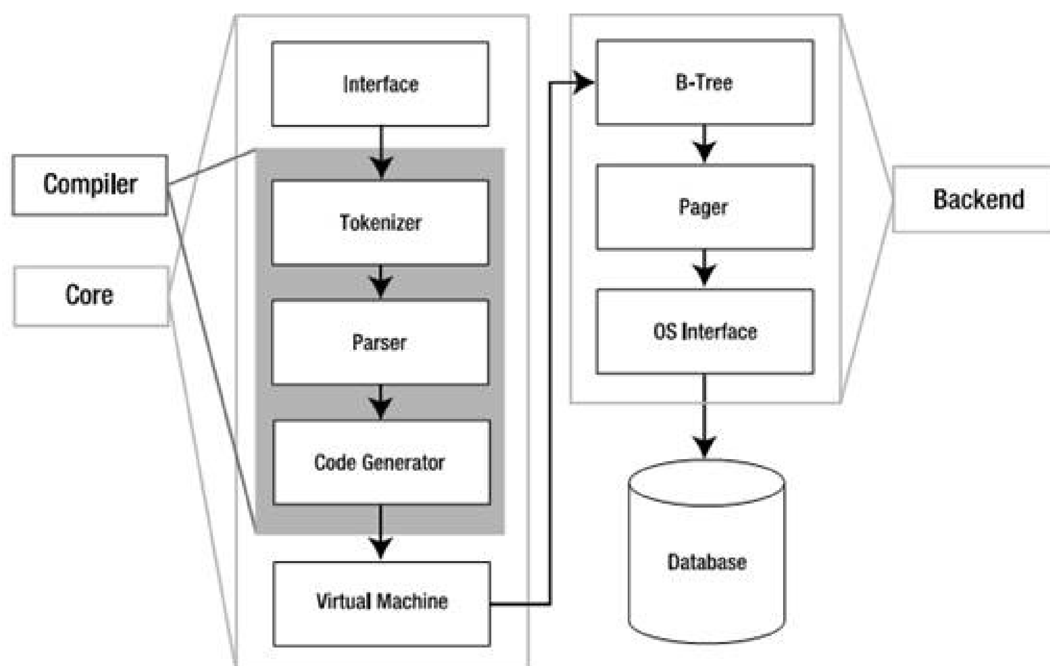


图1-2 SQLite的体系结构

接口(Interface)

接口由SQLite C API组成，也就是说不管是程序、脚本语言还是库文件，最终都是通过它与SQLite交互的(我们经常使用的ODBC/JDBC最后也会转化为相应C API的调用)。

编译器(Compiler)

编译过程从分词器(Tokenizer)和分析器(Parser)开始。它们协作处理文本形式的结构化查询(Structured Query Language, SQL)语句，分析其语法有效性，转化为底层能更方便处理的层次数据结构——语法树，然后把语法树传给代码生成器(code generator)进行处理。SQLite分词器的代码是手工编写的，分析器代码是由SQLite定制的分析器生成器(称为Lemon)生成的。The Lemon parser generator is designed for high performance and takes special precautions to guard against memory leaks. 一旦SQL语句被分解为串值并组织到语法树中，分析器就将该树下传给代码生成器进行处理。而代码生成器根据它生成一种SQLite专用的汇编代码，最后由虚拟机(Virtual Machine)执行。

虚拟机(Virtual Machine)

架构中最核心的部分是虚拟机，或者叫做虚拟数据库引擎(Virtual DataBase Engine,VDBE)。它和Java虚拟机相似，解释执行字节代码。VDBE的字节代码(称为虚拟机语言)由128个操作码(opcodes)构成，主要是进行数据库操作。它的每一条指令或者用来完成特定的数据库操作(比如打开一个表的游标、开始一个事务等)，或者为完成这些操作做准备。总之，所有的这些指令都是为了满足SQL命令的要求。VDBE的指令集能满足任何复杂SQL命令的要求。所有的

SQLite SQL语句——从选择和修改记录到创建表、视图和索引——都是首先编译成此种虚拟机语言，组成一个独立程序，定义如何完成给定的命令。例如，在SQLite的CLP中执行下面语句：

```
sqlite> .m col
sqlite> .h on
sqlite> .w 4 15 3 3 3 10 3
sqlite> explain SELECT name FROM episodes LIMIT 10;
SQLite会显示编译后的VDBE汇编程序，如列表1-1所示。
列表1-1 VDBE汇编程序
addr  opcode           p1   p2   p3   p4               p5    comment
----  ---------------  ---  ---  ---  ---------------  ----  ----------
0     Trace            0    0    0                     00
1     Integer          10   1    0                     00
2     MustBeInt        1    0    0                     00
3     IfZero           1    13   0                     00
4     Goto             0    14   0                     00
5     OpenRead         0    2    0    3                00
6     Rewind           0    12   0                     00
7     Column           0    2    2                     00
8     ResultRow        2    1    0                     00
9     AddImm           1    -1   0                     00
10    IfZero           1    12   0                     00
11    Next             0    7    0                     01
12    Close            0    0    0                     00
13    Halt             0    0    0                     00
14    Transaction      0    0    0                     00
15    VerifyCookie     0    40   0                     00
16    TableLock        0    2    0    episodes         00
17    Goto             0    5    0                     00
```

程序由17条指令组成。通过对给定的操作数完成特别的操作，这些指令将会返回episodes表前10个记录的name字段的值。episodes表是本书示例数据库的一部分。

从多个方面都可以看出，VDBE是SQLite的核心：它上面的各模块都是用于创建VDBE程序，它下面的各模块都是用于执行VDBE程序，每次执行一条指令。

后端(Back-end)

后端由B-tree、页缓冲(page cache，pager)和操作系统接口(即系统调用)构成。B-tree和page cache共同对数据进行管理。它们操作的是数据库页，这些页具有相同的大小，就像集装箱。页里面的“货物”是表示信息的大量bit，这些信息包括记录、字段和索引入口等。B-tree和pager都不知道信息的具体内容，它们只负责“运输”这些页，页不关心这些“集装箱”里面是什么。

B-tree的主要功能就是索引，它维护着各个页之间的复杂的关系，便于快速找到所需数据。它把页组织成树型的结构(这是它名称的由来)，这种树是为查询而高度优化了的。Page为B-tree服务，为它提供页 。Pager的主要作用就是通过OS接口在B-tree和磁盘之间传递页 。磁盘操作是计算机到目前为止所必须做的最慢的事情。所以，pager 尽力提高速度，其方法是把经常使用的页存放到内存当中的页缓冲区里，从而尽量减少操作磁盘的次数。它使用特殊的算法来预测下面要使用哪些页，从而使B-tree能够更快地工作。

工具和测试代码(Utilities and Test Code)

工具模块中包含各种各样的实用功能，还有一些如内存分配、字符串比较、Unicode转换之类的公共服务也在工具模块中。这个模块就是一个包罗万象的工具箱，很多其它模块都需要调用和共享它。

测试模块中包含了无数的回归测试语句，用来检查数据库代码的每个细微角落。这个模块是SQLite性能如此可靠的原因之一。

SQLite 的特色

尽管SQLite是如此之小，却提供了如此之多的特色和性能。它支持ANSI SQL92的一个大子集(包括事务、视图、检查约束、关联子查询和复合查询等)，还支持其它很多关系型数据库的特色，如触发器、索引、自动增长字段和LIMIT/OFFSET子句等。SQLite还有很多独特的特色，如内在数据库、动态类型和冲突解决(下面解释)。

如本章开始时所述，在SQLite的观念和实现中，都遵循着一系列指导原则。下面就来详述这些原则。

零配置

从SQLite的设计之始，就没准备在应用时使用DBA。配置和管理SQLite就像得到它一样简单。SQLite包含了正好适合于一个程序员的脑筋的特色。

兼容性

SQLite在设计时特别注意了兼容性。它可以编译运行在Windows、Linux、BSD、Mac OS X及商用的Unix 系统如Solaris、HPUX和AIX，还可以应用于很多嵌入式平台如QNX、VxWorks、Symbian、Palm OS和Windows CE。它可以无缝地工作在16-bit、32-bit和64-bit体系结构中并且能同时适应字节的大端格式和小端格式。SQLite的兼容性并不只表现在代码上，还表现在其数据库文件上。SQLite的数据库文件在其所支持的所有操作系统、硬件体系结构和字节顺序上都是二进制一致的。你可以在Sun SPARC工作站上创建一个SQLite数据库然后在Mac或Windows的机器上——甚至移动电话上——使用它，而不需要做任何转换和修改。此外，SQLite数据库可以支撑2TB的数据量(受操作系统限制)，还内置地同时支持UTF-8和UTF-16编码。

紧凑性

SQLite的设计可以说是功能齐全但体积很小：1个头文件，1个库，不需要扩展的数据库服务。所有的东西，包括客户端、服务器和虚拟机等，都被打包在1/4兆大小之内。如果在编译时去掉一些不需要的特性，程序库可以缩小至170KB (在x86硬件平台上使用GNU C进行编译)。此外，还有一个SQLite的私有版本，大小是69KB，可以运行在智能卡上(参“附加信息”一节)。

空注：我下载的DLL有500多KB。

简单

作为程序库，SQLite的API可以算是最简单最易用的了。SQLite既有很好的文档又很容易望文知意。

适应性

SQLite的几个特性使其成为一个适应性极强的数据库。作为一个内嵌式的数据库，SQLite在以下两个方面都做得最好：强有力而可伸缩的关系型数据库前端，简单而紧凑的B-tree后端。

不受拘束的授权

SQLite的全部代码都在公共域中，不需要授权。SQLite的任何一部分都没有附加版权要求。所有曾经为SQLite项目贡献过代码的人都签署过一个宣誓书将他们的贡献发布到公共域。也就是说，无论你如何使用SQLite的代码都不会有法律方面的限制。你可以修改、合并、发布、出售或将这些代码用于任何目的，商业和中非商业的，不需要支付任何费用，不会受到任何限制。

可靠性

SQLite的源代码不但免费，还编写得很好。SQLite源代码包含大约30000行标准C代码，它是干净的、模块化的和完好注释的。SQLite源代码易理解、易定制。

SQLite的核心软件(库和工具)由约30000行代码组成，但分发的程序中还包含有超过30000行的回归测试代码，它们覆盖了97%的核心代码。也就是说，超过一半的SQLite项目代码是专门用于回归测试的，也就是说，差不多每写一行功能代码，都要写一行测试代码对它进行测试。

易用性

SQLite还提供一些独特的功能来提高易用性，包括动态类型、冲突解决和“附加”多个数据库到一个连接的能力。

性能和限制

SQLite是一个快速数据库。但“快速”这个词本身是一个主观的和不明确的词。诚实地讲，有些事情SQLite能比其它数据库做得快，也有些事情不能。这么说吧，利用SQLite提供的配置参数，SQLite是足够快速和高效的。跟大多数其它数据库一样，SQLite使用B-tree处理索引，使用B+tree处理表数据。因此，在对单表进行查询时，SQLite要快于(或至少相当于)其它数据库的速度。

在一些情况下SQLite可能不如大型数据库快，但大多数这些情况是可理解的。SQLite是一个内嵌式的数据库，设计用于中小规模的应用程序。这些限制是符合设计目的的。很多新用户错误地假设使用SQLite可以代替大型关系型数据库，这有时行，但有时不行，依赖于你准备用SQLite来做什么。一般情况下，SQLite在三个主要的方面具有局限性：

- 并发。
- 数据库大小。
- 网络。

尽管SQLite做得已经很好了，但仍有部分特性未能实现，包括：

- 外键约束 空注：SQLite的最新版本3.6.19好像已经支持了。
- 完整的触发器支持。
- 完整的ALTER TABLE支持。
- 事务嵌套。
- RIGHT和FULL OUTER JOIN。
- 可修改视图。
- GRANT和REVOKE。

附加信息

SQLite网站有丰富的信息，包括官方文档、邮件列表、Wiki和其它的一般信息，它的网址是www.sqlite.org。SQLite社区也是很有帮助的，你可能从邮件列表中找到任何你所需要的东西。另外，SQLite的作者提供了SQLite的专业培训和支持，包括定制程序(如移植到嵌入式平台)和增强的SQLite版本，这些版本包括内置了加密功能的版本和为嵌入式应用优化的极小化版本。更多的信息可以从www.hwaci.com/sw/sqlite/prosupport.html中找到。

第2章 入门

无论您使用何种操作系统，SQLite都很容易上手。对大多数用户，安装SQLite并创建一个新的数据库不会超过5分钟，且不需要任何经验。

空注：本章我只看了Windows操作系统下使用VC的内容。

从哪得到SQLite

SQLite网站(www.sqlite.org)同时提供SQLite的已编译版本和源程序。编译版本可同时适用于Windows和Linux。

有几种形式的二进制包供选择，以适应SQLite的不同使用方式。包括：

- 静态链接的命令行程序(CLP)
- SQLite动态链接库(DLL)
- Tcl扩展

SQLite源代码以两种形式提供，以适应不同的平台。一种为了在Windows下编译，另一种为了在POSIX平台(如Linux, BSD, and Solaris)下编译，这两种形式下源代码本身是没有差别的。

在Windows上使用SQLite

无论你是作为终端用户还是作为程序员来使用SQLite，SQLite都可以很容易地安装在Windows环境下。本节我们将讨论所有相关的内容——安装二进制包或在最普通的编译环境下使用源代码。

获得命令行程序

SQLite命令行程序(CLP)是开始使用SQLite的一个比较好的选择。

略，参原文。

获得SQLite的动态链接库(DLL)

SQLite的DLL文件供编译好的程序动态连接SQLite。大多数使用SQLite的软件都会拥有自己的SQLite DLL拷贝并随软件自动安装。

在Windows环境下编译SQLite源代码

在Windows环境下编译SQLite源代码是很简单的。根据你所使用的编译器和你要做什么，有几种方法来编译SQLite。最常见的环境是Microsoft Visual C++或MinGW，本节都会加以介绍。关于使用其它编译器编译SQLite的内容，可参考SQLite Wiki (www.sqlite.org/cvstrac/wiki?p=HowToCompile)。

用Microsoft Visual C++构建SQLite DLL

通过以下步聚，可使用源代码，在Visual C++上构建SQLite DLL：

1. 启动Visual Studio。在解包的SQLite源程序目录中创建一个新的DLL“空”项目。

高：不同版本操作略有不同，不详细解释了。

2. 将全部SQLite源文件加入到项目中来。包括所有的.c文件和.h文件。除了：

shell.c：该文件包括main()函数，用于创建CLP可执行程序。

tclsqlite.c：该文件用于TCL支持。

空注：我使用的版本(sqlite-source-3_6_18.zip)有些函数有重复定义，还得去掉两个文件，不知会引起什么后果，它们是fts3.c和fts3_tokenizer.c。

3. 执行构建(Build)命令，OK。

还可以选择构建线程完全的DLL或发布 (Release) 版的DLL，参原文。

用Microsoft Visual C++构建SQLite CLP

方法基本同上。

创建项目时选择Win32 Console Application，添加文件时把shell.c也加上，即可。

使用SQLite数据库

SQLite的CLP是使用和管理SQLite数据库最常用的方法。

它可运行于多种平台，学会使用CLP，可以保证你永远有一个通用和熟悉的途径来管理你的数据库。CLP其实是两个程序。它可以运行在命令行模式下完成各种数据库管理任务，也可以运行在Shell模式下，以交互的方式执行查询操作。

Shell模式下使用CLP

运行DOS shell，进入工作目录，在命令行上键入sqlite3命令，命令后跟随一个可选的数据库文件名。如果在命令行上不指定数据库名，SQLite将会使用一个内存数据库，其内容在退出CLP时将会丢失。

CLP以交互形式运行，你可以在其上执行查询、获得schema信息、导入/导出数据和执行其它各种各样的数据库任务。CLP认为你输入的任何语句都是一个查询命令(query)，除非命令是以点(.)开始，这些命令用于特殊操作。键入.help或.h可以得到这些操作的完整列表。键入.exit或.e退出CLP。

让我们从创建一个称为test.db的数据库开始。在DOS shell下键入：

sqlite3 test.db

尽管我们提供了数据库名，但如果这个数据库并不存在，SQLite并不会眞正地创建它。SQLite会等到你眞正地向其中增加了数据库对象之后才创建它，比如在其中创建了表或视图。这样做的原因是给你机会在将数据库写到外部文件之前对数据库做一些永久性的设置，如页的大小等。有些设置，如页大小、字符集(UTF-8或UTF-16)等，一旦数据库创建之后就不能再修改了。这个中间期是你能改它们的唯一机会。我们采用默认设置，因此，要将数据库写到磁盘，我们仅需要在其中创建一个表。输入如下语句：

sqlite> create table test (id integer primary key, value text);

现在你有了一个称为test.db的数据库文件，其中包含一个表test，该表包含两个字段。

- 一个称为id的主键字段,它带有自动增长属性。无论何时你定义一个整型主键字段，SQLite 都会对该字段应用自动增长属性。
- 一个简单的称为value的文本字段。

向表中插入几行数据：

```
sqlite> insert into test (value) values('eenie');
sqlite> insert into test (value) values('meenie');
sqlite> insert into test (value) values('miny');
sqlite> insert into test (value) values('mo');
```

将插入的数据取回：

```
sqlite> .mode col
sqlite> .headers on
sqlite> SELECT * FROM test;
```

系统显示：

```
id value
1 eenie
2 meenie
3 miny
4 mo
```

SELECT语句前的两个命令(.headers and .mode)用于改进输出的格式。可以看到SQLite为id字段赋予了连接的整数值，而这些值我们在INSERT语句中并没的提供。对于自动增长的字段，你可能会关心最后插入的一条记录该字段的取值，此值可以用SQL函数last_insert_rowid()得到。

```
sqlite> select last_insert_rowid();
last_insert_rowid()
4
```

在退出CLP之前，让我们来为数据库创建一个索引和一个视图，后面的内容中将会用到它们。

```
sqlite> create index test_idx on test (value);
sqlite> create view schema as select * from sqlite_master;
```

使用.exit命令退出CLP。

```
sqlite> .exit
C:\Temp>
```

获得数据库的Schema信息

有几个shell命令用于获得有关数据库内容的信息。你可以键入命令.tables [pattern]来得到所有表和视图的列表，其中[pattern]可以是任何类SQL的操作符。执行上述命令会返回符合条件的所有表和视图，如果没有pattern项，返回所有表和视图。

```
sqlite> .tables
schema test
```

可以看到我们创建的表test和视图schema。同样的，要显示一个表的索引，可以键入命令.indices [table name]：

```
sqlite> .indices test
test_idx
```

可以看到我们为表test所创建的名为test_idx的索引。使用.schema [table name]可以得到一个表或视图的定义(DDL)语句。如果没提供表名，则返回所有数据库对象(包括table、indexe、view和index)的定义语句：

```
sqlite> .schema test
CREATE TABLE test (id integer primary key, value text);
CREATE INDEX test_idx on test (value);
sqlite> .schema
CREATE TABLE test (id integer primary key, value text);
CREATE VIEW schema as select * from sqlite_master;
CREATE INDEX test_idx on test (value);
```

更详细的schema信息可以通过SQLite唯一的一个系统视图sqlite_master得到。这个视图是一个系统目录，它的结构如表2-1所示。

表2-1 sqlite_master表结构

编号	字段	说明
1	type	值为"table"、 "index"、 "trigger"或"view"之一。
2	name	对象名称，值为字符串。
3	tbl_name	如果是表或视图对象，此字段值与字段2相同。如果是索引或触发器对象，此字段值为与其相关的表名。
4	rootpage	对触发器或视图对象，此字段值为0。对表或索引对象，此字段值为其根页的编号。
5	SQL	字符串，创建此对象时所使用的SQL语句。

查询当前数据库的sqlite_master表，返回：

```
sqlite> .mode col
sqlite> .headers on
sqlite> select type, name, tbl_name, sql from sqlite_master order by type;
type   name       tbl_name   sql
---------
index  test_idx   test       CREATE INDEX test_idx on test (value)
table  test       test       CREATE TABLE test (id integer primary
view   schema     schema     CREATE VIEW schema as select * from s
```

■Tip：使用向上的箭头键可以回滚到前面输入过的命令。

数据导出

可以使用.dump命令将数据库导出为SQL格式的文件。不使用任何参数，.dump将导出整个数据库。如果提供参数，CLP把参数理解为表名或视图名。

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .output stdout
```

数据导入

有两种方法可以导入数据，用哪种方法决定于要导入的文件的格式。如果文件由SQL语句构成，可以使用.read命令导入(执行)文件。如果文件是由逗号或其它定界符分隔的值(comma-separated values, CSV)组成，可使用.import [file][table]命令。此命令将解析指定的文件并尝试将数据插入到指定的表中。

```
sqlite> .show
echo: off
explain: off
headers: on
mode: column
nullvalue: ""
output: stdout
separator: "|"
width:
```

.read命令用来导入由.dump命令创建的文件。如果要使用前面作为备份文件所导出的file.sql，需要先移除已经存在的数据库对象(test表和schema视图)，然后用下面方法导入：

```
sqlite> drop table test;
sqlite> drop view schema;
sqlite> .read file.sql
```

格式化

CLP提供了几个格式化选项命令。最简单的是.echo, 如果设置.echo on，则新输入的命令在执行前都会回显，默认值是off。.headers设置为on时，查询结果显示时带有字段名。当遇到NULL值时，如果需要以一个字符串来显示，使用.nullvalue命令设置，如：

```
sqlite> .nullvalue NULL
```

默认情况下使用空串。如果要改变CLP的shell提示符，使用.prompt [value]，如：

```
sqlite> .prompt 'sqlite3> '
sqlite3>
```

.mode命令可以设置结果数据的几种输出格式。可选的格式为csv、column、html、insert、line、list、tabs和tcl。默认值是list，在此模式下显示结果时列间以默认的分隔符分隔。如果你想以CSV格式输出一个表的数据，可如下操作：

```
sqlite3> .output file.csv
sqlite3> .separator ,
sqlite3> select * from test;
sqlite3> .output stdout
```

文件file.csv的内容为：

```
1,eenie
2,meenie
3,miny
4,mo
```

因为有一个CSV模式，所以下面的命令会得到相似的结果：

```
sqlite3> .output file.csv
sqlite3> .mode csv
sqlite3> select * from test;
sqlite3> .output stdout
```

在命令行方式下执行CLP

在DOS或UNIX的命令行方式下，直接执行SQLite的数据库操作。

数据库管理

所有的数据库管理任务都可以在shell和命令行模式下完成。

创建、备份和删除数据库

数据库的备份有两种方法。第1种是使用.dump，可得到SQL格式的文件。在命令行方式下可如下做：

```
sqlite3 test.db .dump > test.sql
```

在CLP中可如下做：

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .exit
```

相应地，导入一个SQL格式备份的数据库可如下做：

```
sqlite3 test.db < test.sql
```

此处假设test.db不存在。如果它存在，则或许会因为数据库中有同名的对象而出错。

可以用复制的方法得到一个二进制的数据库文件拷贝。但也许在复制之前你想先抽空(vacuum)它，也就是释放数据库文件中未使用的空间，以得到一个更小的数据库文件。可操作如下：

```
sqlite3 test.db VACUUM
cp test.db test.backup
```

一般情况下，二进制的备份如不SQL备份兼容性好。尽管SQLite有很好的向上兼容性和各操作系统间文件格式的一致性，但如果想要将备份文件保留很长时间，还是SQL格式保险一些。

当一个数据库你不想再用时，简单地从操作系统中将其文件删除就行了。

获得数据库文件的信息

按前文所述，获得数据库信息的主要途径是使用sqlite_master视图，它提供一个数据库所包含的所有对象的细节信息。

如果你想获得关于物理的数据库结构信息，可以使用一个称为SQLite Analyzer的工具，它可以在SQLite网站上下载得到。SQLite Analyzer可以提供磁盘SQLite数据库的详细技术信息。

(输出结果略)

其它SQLite工具

有很多其它开源的或商业的程序可工作于SQLite，其中具有优秀图形化界面且跨平台的有：

- SQLite Database Browser (http://sqlitebrowser.sourceforge.net)
- SQLite Control Center (http://bobmanc.home.comcast.net/sqlitecc.html)
- SQLiteManager (www.sqlabs.net/sqlitemanager.php)

第3章 关系模型

SQL具有非常实用的外观和非常理论化的内涵，这个内涵就是关系模型。关系模型早于SQL出现并对SQL的出现提出了需求。SQL的原动力不在语言本身，而是深藏在关系模型的概念当中。这些概念构成了SQL设计和操作的基础。

空注：数据库基本理论，参考其它书吧。

第4章 SQL

本章介绍SQL的基本内容和SQLite的特殊实现。本章内容的编排假设你没有SQL和关系模型的基础知识。如果你是SQL新手，SQLite将带你进入关系模型的精彩世界。

空注：使用过很多种数据库，所以本章只关注SQLite与其它DBMS不同的地方，如弱类型什么的。

关系模型

如第3章所述，SQL是关系模型的产物，关系模型是由E. F. Codd在1969年提出的。关系模型要求关系型数据库能够提供一种查询语言，几年后，SQL应运而生。

关系模型由三部分构成：表单(form)、功能(function)和一致性(consistency)。表单表示信息的结构。在关系模型中只使用一种单独的数据结构来表达所有信息，这种结构称为关系(relation，在SQL中被称为表、table)。关系由多个元组(tuples，在SQL中被称为行、记录、rows)构成，每个元组又由多个属性(attributes，在SQL中被称为列、字段、columns)构成。

查询语言

查询语言将外部世界和数据的逻辑表现联系在一起，并使它们能够交互。它提供了取数据和修改数据的途径，是关系模型的动态部分。

SQL的发展

第一个被采用的此类查询语言可能是在IBM的System R当中。System R是一个关系型数据库的研究项目，此项目直接派生出了Codd的论文。这个语言开始时被称作SEQUEL，是“Structured English Query Language”的缩写。后来被缩短为SQL，或“Structured Query Language”。

示例数据库

示例数据库在本章和后面的章节中将会用到，其中存储了Seinfeld所有episode(约180个)的食品(约412种)。数据库中的表如图4-1所示。

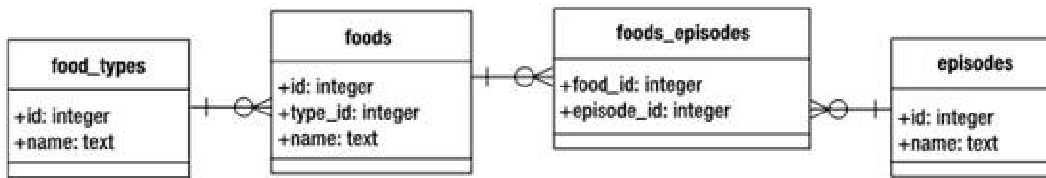


图4-1 Seinfeld食品数据库

数据库的schema定义如下：

```
create table episodes (
  id integer primary key,
  season int,
  name text );
create table foods(
  id integer primary key,
  type_id integer,
  name text );
create table food_types(
  id integer primary key,
  name text );
  create table foods_episodes(
  food_id integer,
  episode_id integer );
```

主表是foods。foods中的每条记录代表一种食品，其名称存储于name字段。type_id参照food_types，food_types表存储了食品的分类(如烘烤食品、饮品、垃圾食品等)。foods_episodes表是foods和episodes的关联表。

建立

示例数据库文件可在随书的zip文件中找到。

运行示例

为了您的方便，本章的所有SQL示例都保存在随书zip文件根目录的sql.sql文件中。

对于长SQL语句，一个方便的执行方法是将其复制到一个单独的文本文件，如test.sql中，然后执行：

```
sqlite3 foods.db < test.sql
```

为了增加输出的易读性，你应用把下面内容包含在文件中：

```
.echo on
.mode col
.headers on
.nullvalue NULL
```

语法

SQL的语法很像自然语言。每个语句都是一个祈使句，以动词开头，表示所要做的动作。后面跟的是主题和谓词，如图4-2所示。

```
select id from foods where name='JujyFruit';
|____||_________||______________________|
 verb   subject         predicate
```

图 4-2 一般的SQL语法结构

命令

SQL由命令组成，每个命令以分号(;)结束。如下面是3个独立的命令：

```
SELECT id, name FROM foods;
INSERT INTO foods VALUES (NULL, 'Whataburger');
DELETE FROM foods WHERE id=413;
```

常量

也称为Literals，表示确切的值，有3种：字符串常量、数据常量和二进制常量。字符串常量如：

```
'Jerry'
'Newman'
'JujyFruit'
```

字符串值用单引号(')括起来，如果字符串中本身包含单引号，需要双写。如“Kenny’s chicken”需要写成：

```
'Kenny''s chicken'
```

数字常量有整数、十进制数和科学记数法表示的数，如：

```
-1
3.142
6.0221415E23
```

二进制值用如x'0000'的表示法，其中每个数据是一个16进制数。二进制值必须由两个两个的16进制数(8 bits)组成，如：

```
x'01'
X'0fff'
x'0F0EFF'
X'0f0effab'
```

保留字和标识符

保留字由SQL保留用做特殊的用途，如SELECT、UPDATE、INSERT、CREATE、DROP和BEGIN等。标识符指明数据库里的具体对象，如表或索引。保留字预定义，不能用做标识符。SQL不区分大小写，下面是相同的语句：

```
SELECT * from foo;
SeLeCt * FrOm FOO;
```

为清楚起见，本章中保留字都用大写，标识符都用小写。

但是，SQLite对字符串的值是大小写敏感的。

注释

SQL中单行注释用双减号开始，多行注释采用C风格的/ /形式。

创建一个数据库

数据库中所有的工作都围绕表进行。表由行和列组成，看起来简单，但其实并非如此。表跟其它所有的概念有关，涉及本章的大部分篇幅。在此我们用2分钟的时间给出一个预览。

创建表

在SQL中，创建和删除数据库对象的语句一般被称为数据定义语言(data definition language， DDL)，操作这些对象中数据的语句称为数据操作语言(data manipulation language，DML)。创建表的语句属于DDL，用CREATE TABLE命令，如下定义：

```
CREATE [TEMP] TABLE table_name (column_definitions [, constraints]);
```

用TEMP或TEMPORARY保留字声明的表为临时表，只存活于当前会话，一旦连接断开，就会被自动删除。

中括号表示可选项。

另外，竖线表示在多个中选一，如：

```
CREATE [TEMP|TEMPORARY] TABLE … ;
```

如果没有指明创建临时表，则创建的是基本表，将会在数据库中持久存在。

数据库中还有其它类型的表，如系统表和视图，现在先不介绍。

CREATE TABLE命令至少需要一个表名和一个字段名。命令中table_name表示表名，必须与其它所有的标识符不同。column_definitions表示一个用逗号分隔的字段列表。每个字段定义包括一个名称、一个域和一个逗号分隔的字段约束表。“域”一般情况下是一个类型，与编程语言中的数据类型同名，指明存储在该列的数据的类型。在SQLite中有5种本地类型：INTEGER、REAL、TEXT、BLOB和NULL，所有这些域将在本章后面的“存储类”一节中介绍。“约束”用来控制什么样的值可以存储在表中或特定的字段中。例如，你可以用UNIQUE约束来规定所有记录中某个字段的值要各不相同。约束将会在“数据完整性”一节中介绍。

在字段列表后面，可以跟随一个附加的字段约束，如下例：

```
CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
    name TEXT NOT NULL COLLATE NOCASE,
    phone TEXT NOT NULL DEFAULT 'UNKNOWN',
    UNIQUE (name,phone) );
```

改变表

你可以用ALTER TABLE命令改变表的结构。SQLite版的ALTER TABLE命令既可以改变表名，也可以增加字段。一般格式为：

```
ALTER TABLE table { RENAME TO name | ADD COLUMN column_def }
```

注意这里又出现了新的符号{}。花括号括起来一个选项列表，必须从各选项中选择一个。此处，我们或者ALTER TABLE table RENAME…，或者ALTERTABLE table ADD COLUMN…。That is, you can either rename the table using the RENAME clause, or add a column with the ADDCOLUMN clause. To rename a table, you simply provide the new name given by name. If you add a column, the column definition, denoted by column_def, follows the form in the CREATE TABLE statement. It is a name, followed by an optional domain and list of constraints. 例如：

```
sqlite> ALTER TABLE contacts
    ADD COLUMN email TEXT NOT NULL DEFAULT '' COLLATE NOCASE;
sqlite> .schema contacts
    CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
    name TEXT NOT NULL COLLATE NOCASE,
    phone TEXT NOT NULL DEFAULT 'UNKNOWN',
    email TEXT NOT NULL DEFAULT '' COLLATE NOCASE,
    UNIQUE (name,phone) );
```

显示了当前的表定义。

表还可以由SELECT语句创建，你可以在创建表结构的同时创建数据。这种特别的CREATE TABLE语句将在“插入记录”一节中介绍。

在数据库中查询

SELECT是SQL命令中最大最复杂的命令。SELECT的很多操作都来源于关系代数。

关系操作

SELECT中使用3大类13种关系操作：

- 基本的操作
  - Restriction(限制)
  - Projection
  - Cartesian Product(笛卡尔积)
  - Union(联合)
  - Difference(差)
  - Rename(重命名)
- 附加的操作
  - Intersection(交叉)
  - Natural Join(自然连接)
  - Assign(指派 OR 赋值)
- 扩展的操作
  - Generalized Projection
  - Left Outer Join
  - Right Outer Join
  - Full Outer Join

基本的关系操作，除重命名外，在集合论中都有相应的理论基础。附加操作是为了方便， 它们可以用基本操作来完成，一般情况下，附加操作可以作为常用基本操作序列的快捷方式。扩展操作为基本操作和附加操作增加特性。

ANSI SQL的SELECT可以完成上述所有的关系操作。这些操作覆盖了Codd最初定义的所有关系运算符，只有一个例外——divide。SQLite支持ANSI SQL中除right和full outer join之外的所有操作(这些操作可用其它间接的方法完成)。

操作管道

从语法上来说，SELECT命令用一系列子句将很多关系操作组合在一起。每个子句代表一种特定的关系操作。几乎所有这些子句都是可选的，你可以只选你所需要的操作。

SELECT是一个很大的命令。下面是SELECT的一个简单形式：

```
SELECT DISTINCT heading FROM tables WHERE predicate
GROUP BY columns HAVING predicate
ORDER BY columns LIMIT count,offset;
```

每个保留字——DISTINCT、FROM、WHERE和HAVING——都是一个单独的子句。每个子句由保留字和跟随的参数构成。

表4-1 SELECT的子句

编号	子句	操作	输入
1	FROM	Join	List of tables
2	WHERE	Restriction	Logical predicate
3	ORDER BY	List of columns	
4	GROUP BY	Restriction	List of columns
5	HAVING	Restriction	Logical predicate
6	SELECT	Restriction	List of columns or expressions
7	DISTINCT	Restriction	List of columns
8	LIMIT	Restriction	Integer value
9	OFFSET	Restriction	Integer value

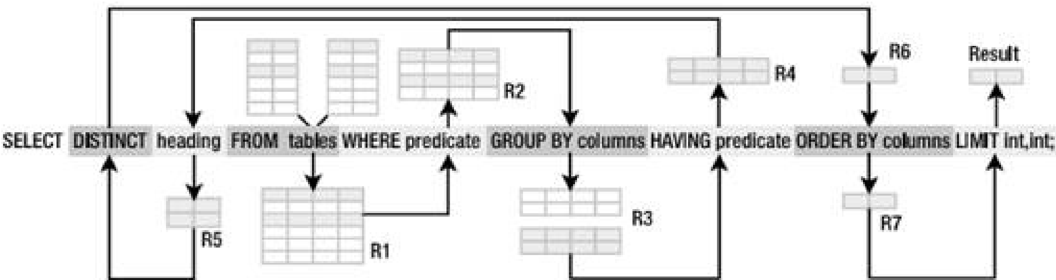


图4-3 SELECT phases

过滤

如果SELECT是SQL中最复杂的命令，那么WHERE就是SELECT中最复杂的子句。

值

“值”可以按它们所属的域(或类型)来分类，如数字值(1, 2, 3, etc.)或字符串值(“Jujy-Fruit”)。值可以表现为文字的值(1, 2, 3 or “JujyFruit”)、变量(一般是如foods.name的列名)、表达式(3+2/5)或函数的结果(COUNT(foods.name))值。

操作符

操作符使用一个或多个值做为输入并产生一个新值做为输出。这所以叫“操作符”是因为它完成某种操作并产生某种结果。二目操作符操作两个输入值(或称操作数)，三目操作符操作三个操作数，单目操作符操作一个操作数，等等。

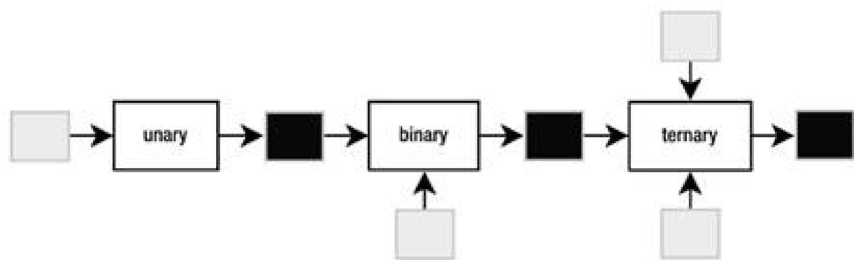


图4-7 单目、二目和三目操作符

二目操作符

二目操作符是最常用的SQL操作符。表4-2列出了SQLite所支持的二目操作符。表中按优先级从高到低的次序排列，同色的一组中具有相同的优先级，圆括号可以覆盖原有的优先级。

表4-2二目操作符

操作符	类型	作用
||	String	Concatenation
*	Arithmetic	Multiply
/	Arithmetic	Divide
%	Arithmetic	Modulus
+	Arithmetic	Add
–	Arithmetic	Subtract
<<	Bitwise	Right shift
>>	Bitwise	Left shift
&	Logical	And
|	Logical	Or
<	Relational	Less than
<=	Relational	Less than or equal to
>	Relational	Greater than
>=	Relational	Greater than or equal to
=	Relational	Equal to
==	Relational	Equal to
<>	Relational	Not equal to
!=	Relational	Not equal to
IN	Logical	In
AND	Logical	And
OR	Logical	Or
LIKE	Relational	String matching
GLOB	Relational	Filename matching

LIKE操作符

一个很有用的关系操作符是LIKE。LIKE的作用与相等(=)很像，但却是通过一个模板来进行字符串匹配。例如，要查询所有名称以字符“J”开始的食品，可使用如下语句：

```
sqlite> SELECT id, name FROM foods WHERE name LIKE 'J%';
id name
156 Juice box
236 Juicy Fruit Gum
243 Jello with Bananas
244 JujyFruit
245 Junior Mints
370 Jambalaya
```

模板中的百分号(%)可与任意0到多个字符匹配。下划线(_)可与任意单个字符匹配。

```
sqlite> SELECT id, name FROM foods WHERE name LIKE '%ac%P%';
id name
127 Guacamole Dip
168 Peach Schnapps
198 Mackinaw Peaches
```

另一个有用的窍门是使用NOT：

```
sqlite> SELECT id, name FROM foods
WHERE name like '%ac%P%' AND name NOT LIKE '%Sch%'
id name
38 Pie (Blackberry) Pie
127 Guacamole Dip
198 Mackinaw peaches
```

限定和排序

可以用LIMIT和OFFSET保留字限定结果集的大小和范围。LIMIT指定返回记录的最大数量。OFFSET指定偏移的记录数。例如，下面的命令返回food_types表中id排第2的记录：

```
SELECT * FROM food_types LIMIT 1 OFFSET 1 ORDER BY id;
```

保留字OFFSET在结果集中跳过一行(Bakery)，保留字LIMIT限制最多返回一行(Cereal)。

上面语句中还有一个ORDER BY子句，它使记录集在返回之前按一个或多个字段的值排序。例如：

```
sqlite> SELECT * FROM foods WHERE name LIKE 'B%'
ORDER BY type_id DESC, name LIMIT 10;
id type_id name
382 15 Baked Beans
383 15 Baked Potato w/Sour
384 15 Big Salad
385 15 Broccoli
362 14 Bouillabaisse
328 12 BLT
327 12 Bacon Club (no turke
326 12 Bologna
329 12 Brisket Sandwich
274 10 Bacon
```

函数(Function)和聚合(Aggregate)

SQLite提供了多种内置的函数和聚合，可以用在不同的子句中。函数的种类包括：数学函数，如ABS()计算绝对值；字符串格式函数，如UPPER()和LOWER()，它们将字符串的值转化为大写或小写。例如：

```
sqlite&gt; SELECT UPPER('hello newman'), LENGTH('hello newman'), ABS(-12);
UPPER('hello newman') LENGTH('hello newman') ABS(-12)
HELLO NEWMAN 12 12
```

函数名是不分大小写的(或upper()和UPPER()是同一个函数)。函数可以接受字段值作为参数：

```
sqlite> SELECT id, UPPER(name), LENGTH(name) FROM foods
      WHERE type_id=1 LIMIT 10;
id UPPER(name) LENGTH(name)
-----------
1 BAGELS 6
2 BAGELS, RAISIN 14
3 BAVARIAN CREAM PIE 18
4 BEAR CLAWS 10
5 BLACK AND WHITE COOKIES 23
6 BREAD (WITH NUTS) 17
7 BUTTERFINGERS 13
8 CARROT CAKE 11
9 CHIPS AHOY COOKIES 18
10 CHOCOLATE BOBKA 15
```

因为函数可以是任意表达式的一部分，所以函数也可以用在WHERE子句中：

```
sqlite> SELECT id, UPPER(name), LENGTH(name) FROM foods
      WHERE LENGTH(name) < 5 LIMIT 5;
id upper(name) length(name)
36PIE 3
48 BRAN 4
56KIX 3
57 LIFE 4
80 DUCK 4
```

聚合是一类特殊的函数，它从一组记录中计算聚合值。标准的聚合函数包括SUM()、AVG()、COUNT()、MIN()和MAX()。例如，要得到烘烤食品(type_id=1)的数量，可使用如下语句：

```
sqlite> SELECT COUNT(*) FROM foods WHERE type_id=1;
count
47
```

分组(Grouping)

聚合的精华部分是分组。聚合不只是能够计算整个结果集的聚合值，你还可以把结果集分成多个组，然后计算每个组的聚合值。这些都可以在一步当中完成，方法就是使用GROUP BY子句，如：

```
sqlite> SELECT type_id FROM foods GROUP BY type_id;
type_id
1
2
3
.
.
.
15
```

去掉重复

操作管道中的下一个限制是DISTINCT。DISTINCT处理SELECT的结果并过滤掉其中重复的行。例如，你想从foods表中取得所有不同的type_id值：

```
sqlite> SELECT DISTINCT type_id FROM foods;
type_id
1
2
3
.
.
.
15
```

多表连接

连接(join)是SELECT命令的第一个操作，它产生初始的信息，供语句的其它部分过滤和处理。连接的结果是一个合成的关系(或表)，它是SELECT后继操作的输入。

也许从一个例子开始是最简单的。

```
sqlite> SELECT foods.name, food_types.name
      FROM foods, food_types
      WHERE foods.type_id=food_types.id LIMIT 10;
name name
Bagels Bakery
Bagels, raisin Bakery
Bavarian Cream Pie Bakery
Bear Claws Bakery
Black and White cookies Bakery
Bread (with nuts) Bakery
Butterfingers Bakery
Carrot Cake Bakery
Chips Ahoy Cookies Bakery
Chocolate Bobka Bakery
```

名称和别名

当把多个表连接在一起时，字段可能重名。

```
SELECT B.name FROM A JOIN B USING (a);
```

修改数据

跟SELECT命令相比，用于修改数据的语句就太简单太容易理解了。有3个DML语句用于修改数据——INSERT、UPDATE和DELETE。

插入记录

使用INSERT命令向表中插入记录。使用INSERT命令可以一次插入1条记录，也可以使用SELECT命令一次插入多条记录。INSERT语句的一般格式为：

```
INSERT INTO table (column_list) VALUES (value_list);
```

Table指明数据插入到哪个表中。column_list是用逗号分隔的字段名表，这些字段必须是表中存在的。value_list是用逗号分隔的值表，这些值与column_list中的字段一一对应。例如，下面语句向foods表插入数据：

```
sqlite> INSERT INTO foods (name, type_id) VALUES ('Cinnamon Bobka', 1);
```

修改记录

UPDATE命令用于修改一个表中的记录。UPDATE命令可以修改一个表中一行或多行中的一个或多个字段。UPDATE语句的一般格式为：

```
UPDATE table SET update_list WHERE predicate;
```

update_list是一个或多个“字段赋值”的列表，字段赋值的格式为column_name=value。WHERE子句的用法与SELECT语句相同，确定需要进行修改的记录。如：

```
UPDATE foods SET name='CHOCOLATE BOBKA'
    WHERE name='Chocolate Bobka';
SELECT * FROM foods WHERE name LIKE 'CHOCOLATE%';
id type_ name
10 1 CHOCOLATE BOBKA
11 1 Chocolate Eclairs
12 1 Chocolate Cream Pie
222 9 Chocolates, box of
223 9 Chocolate Chip Mint
224 9 Chocolate Covered Cherries
```

删除记录

DELETE用于删除一个表中的记录。DELETE语句的一般格式为：

```
DELETE FROM table WHERE predicate;
```

同样，WHERE子句的用法与SELECT语句相同，确定需要被删除的记录。如：

```
DELETE FROM foods WHERE name='CHOCOLATE BOBKA';
```

数据完整性

数据完整性用于定义和保护表内部或表之间数据的关系。有四种完整性：域完整性、实体完整性、参照完整性和用户定义完整性。

唯一约束

因为唯一(UNIQUE)约束是主键的基础，所以先介绍它。一个唯一约束要求一个字段或一组字段的所有值互不相同，或者说唯一。如果你试图插入一个重复值，或将一个值改成一个已存在的值，数据库将引发一个约束非法，并取消操作。唯一约束可以在字段级或表级定义。

NULL和UNIQUE：

问题：如果一个字段已经声明为UNIQUE，可以向这个字段插入多少个NULL值？

回答：与数据库的种类有关。PostgreSQL和Oracle可以插入多个。Informix和Microsoft SQL Server只能一个。DB2、SQL Anywhere和Borland Inter-Base不能。SQLite采用了与PostgreSQL和Oracle相同的解决方案。

另一个困扰大家的关于NULL的经典问题是：两个NULL值是否相等？你没有足够的信息来证明它们相等，但也没有足够的信息证明它们不等。SQLite的观点是假设所有的NULL都是不同的。所以你可以向唯一字段中插入任意多个NULL值。

主键约束

在SQLite中，当你定义一个表时总要确定一个主键，不管你自己有没有定义。这个字段是一个64-bit整型字段，称为ROWID。它还有两个别名——*ROWID*和OID，用这两个别名同样可以取到它的值。它的默认取值按照增序自动生成。SQLite为主键字段提供自动增长特性。

默认值

保留字DEFAULT为字段提供一个默认值。如果用INSERT语句插入记录时没有为该定做指定值，则为它赋默认值。DEFAULT不是一个约束(constraint)，因为它没有强制任何事情。这所以把它归为域完整性，是因为它提供了处理NULL值的一个策略。如果一个字段没有指定默认址，在插入时也没有为该字段指定值，SQLite将向该字段插入一个NULL。例如，contacts.name字段有一个默认值'UNKNOWN'，请看下面例子：

```
sqlite> INSERT INTO contacts (name) VALUES ('Jerry');
sqlite> SELECT * FROM contacts;
id name phone
Jerry UNKNOWN
```

DEFAULT还可以接受3种预定义格式的ANSI/ISO预定字用于生成日期和时间值。CURRENT_TIME将会生成ANSI/ISO格式(HH:MM:SS)的当前时间。CURRENT_DATE会生成当前日期(格式为YYYY-MM-DD)。CURRENT_TIMESTAMP会生成一个日期时间的组合(格式为YYYY-MM-DD HH:MM:SS)。例如：

```
CREATE TABLE times ( id int,
    date NOT NULL DEFAULT CURRENT_DATE,
    time NOT NULL DEFAULT CURRENT_TIME,
    timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP );
INSERT INTO times(1);
INSERT INTO times(2);
SELECT * FROMS times;
id date time timestamp
1 2006-03-15 23:30:25 2006-03-15 23:30:25
2 2006-03-15 23:30:40 2006-03-15 23:30:40
```

NOT NULL约束

CHECK约束

排序法(Collation)

排序法定义如何唯一地确定文本的值。排序法主要用于规定文本值如何进行比较。不同的排序法有不同的比较方法。例如，某种排序法是大小写不敏感的，于是'JujyFruit'和'JUJYFRUIT'被认为是相等的。另外一个排序法或许是大小写敏感的，这时上面两个字符串就不相等了。

SQLite有3种内置的排序法。默认为BINARY，它使用一个C函数memcmp()来对文本进行逐字节的比较。这很适合于大多数西方语言，如英语。NOCASE对26个字母是大小写不敏感的。Finally there is REVERSE, which is the reverse of the BINARY collation. REVERSE is more for testing (and perhaps illustration) than anything else.

SQLite C API提供了一种创建定制排序法的手段，详见第7章。

存储类(Storage Classes)

如前文所述，SQLite在处理数据类型时与其它的数据库不同。区别在于它所支持的类型以及这些类型是如何存储、比较、强化(enforc)和指派(assign)。下面各节介绍SQLite处理数据类型的独特方法和它与域完整性的关系。

对于数据类型，SQLite的域完整性被称为域亲和性(affinity)更合适。在SQLite中，它被称为类型亲和性(type affinity)。为了理解类型亲和性，你必须先要理解存储类和弱类型(manifest typing)。

SQLite有5个原始的数据类型，被称为存储类。存储类这个词表明了一个值在磁盘上存储的格式，其实就是类型或数据类型的同义词。这5个存储类在表4-6中描述。

表 4-6 SQLite存储类

名称	说明
INTEGER	整数值是全数字(包括正和负)。整数可以是1, 2, 3, 4, 6或 8字节。整数的最大范围(8 bytes)是{-9223372036854775808, 0, +9223372036854775807}。SQLite根据数字的值自动控制整数所占的字节数。空注：参可变长整数的概念。
REAL	实数是10进制的数值。SQLite使用8字节的符点数来存储实数。
TEXT	文本(TEXT)是字符数据。SQLite支持几种字符编码，包括UTF-8和UTF-16。字符串的大小没有限制。
BLOB	二进制大对象(BLOB)是任意类型的数据。BLOB的大小没有限制。
NULL	NULL表示没有值。SQLite具有对NULL的完全支持。

SQLite通过值的表示法来判断其类型，下面就是SQLite的推理方法：

- SQL语句中用单引号或双引号括起来的文字被指派为TEXT。
- 如果文字是未用引号括起来的数据，并且没有小数点和指数，被指派为INTEGER。
- 如果文字是未用引号括起来的数据，并且带有小数点或指数，被指派为REAL。
- 用NULL说明的值被指派为NULL存储类。

- 如果一个值的格式为X'ABCD'，其中ABCD为16进制数字，则该值被指派为BLOB。X前缀大小写皆可。

SQL函数typeof()根据值的表示法返回其存储类。使用这个函数，下面SQL语句返回的结果为：

```
sqlite> select typeof(3.14), typeof('3.14'), typeof(314), typeof(x'3142'), typeof(NULL);
typeof(3.14)   typeof('3.14')  typeof(314)    typeof(x'3142')       typeof(NULL)
real                text                integer           blob                null
```

SQLite单独的一个字段可能包含不同存储类的值。请看下面的示例：

```
sqlite> DROP TABLE domain;
sqlite> CREATE TABLE domain(x);
sqlite> INSERT INTO domain VALUES (3.142);
sqlite> INSERT INTO domain VALUES ('3.142');
sqlite> INSERT INTO domain VALUES (3142);
sqlite> INSERT INTO domain VALUES (x'3142');
sqlite> INSERT INTO domain VALUES (NULL);
sqlite> SELECT ROWID, x, typeof(x) FROM domain;
```

返回结果为：

```
rowid      x            typeof(x)
1            3.142      real
2            3.142      text
3            3142      integer
4            1B          blob
5            NULL     null
```

这带来一些问题。这种字段中的值如何存储和比较？如何对一个包含了INTEGER、REAL、TEXT、BLOB和NULL值的字段排序？一个整数和一个BLOB如何比较？哪个更大？它们能相等吗？

答案是：具有不同存储类的值可以存储在同一个字段中。可以被排序，因为这些值可以相互比较。有完善定义的规则来做这件事。不同存储类的值可以通过它们各自类的“类值”进行排序，定义如下：

1. NULL存储类具有最低的类值。一个具有NULL存储类的值比所有其它值都小(包括其它具有NULL存储类的值)。在NULL值之间，没有特别的可排序值。

2. INTEGER或REAL存储类值高于NULL，它们的类值相等。INTEGER值和REAL值通过其数值进行比较。

3. TEXT存储类的值比INTEGER和REAL高。数值永远比字符串的值低。当两个TEXT值进行比较时，其值大小由“排序法”决定。

4. BLOB存储类具有最高的类值。具有BLOB类的值大于其它所有类的值。BLOB值之间在比较时使用C函数memcmp()。

所以，当SQLite对一个字段进行排序时，首先按存储类排序，然后再进行类内的排序 (NULL类内部各值不必排序) 。下面的SQL说明了存储类值的不同：

```
sqlite> SELECT 3 < 3.142, 3.142 < '3.142', '3.142' < x'3000', x'3000' < x'3001';
```

返回：

```
3 < 3.142              3.142 < '3.142'              '3.142' < x'3000'     x'3000' < x'300
1                    1                           1                           1
```

弱类型(manifest typing)

SQLite使用弱类型。

看下面的表：

```
CREATE TABLE foo( x integer,
    y text, z real );
```

向该表插入一条记录：

```
INSERT INTO foo VALUES ('1', '1', '1');
```

当SQLite创建这条记录时，x、y和z这3个字段中存储的是什么类型呢？答案是INTEGER, TEXT和REAL。

再看下面例子：

```
CREATE TABLE foo(x, y, z);
```

现在执行同样的插入语句：

```
INSERT INTO foo VALUES ('1', '1', '1');
```

现在，x、y和z中存储的是什么类型呢？答案是TEXT、TEXT和TEXT。

那么，是否SQLite的字段类型默认为TEXT呢？再看，还是第2个表，执行如下插入语句：

```
INSERT INTO foo VALUES (1, 1.0, x'10');
```

现在，x、y和z中存储的是什么类型呢？答案是INTEGER、REAL和BLOB。

如果你愿意，可以为SQLite的字段定义类型，这看起来跟其它数据库一样。但这不是必须的，你可以尽管违反类型定义。这是因为在任何情况下，SQLite都可以接受一个值并推断它的类型。

总之，SQLite的弱类型可表示为：1)字段可以有类型，2)类型可以通过值来推断。类型亲和性介绍这两个规定如何相互关联。所谓类型亲和性就是在强类型(strict typing)和动态类型(dynamic typing)之间的平衡艺术。

类型亲和性(Type Affinity)

在SQLite中，字段没有类型或域。当给一个字段声明了类型，该字段实际上仅仅具有了该类型的新和性。声明类型和类型亲和性是两回事。类型亲和性预定SQLite用什么存储类在字段中存储值。在存储一个给定的值时到底SQLite会在该字段中用什么存储类决定于值的存储类和字段亲和性的结合。我们先来介绍一下字段如何获得它的亲和性。

字段类型和亲和性

首先，每个字段都具有一种亲和性。共有四种亲和性：NUMERIC、INTEGER、TEXT和NONE。一个字段的亲和性由它预声明的类型决定。所以，当你为字段声明了类型，从根本上说是为字段指定了亲和性。SQLite按下面的规则为字段指派亲和性：

- 默认的，一个字段默认的亲和性是NUMERIC。如果一个字段不是INTEGER、TEXT或NONE的，那它自动地被指派为NUMERIC亲和性。
- 如果为字段声明的类型中包含了'INT'(无论大小写)，该字段被指派为INTEGER亲和性。
- 如果为字段声明的类型中包含了'CHAR'、'CLOB'或'TEXT'(无论大小写)，该字段被指派为TEXT亲和性。如'VARCHAR'包含了'CHAR'，所以被指派为TEXT亲和性。
- 如果为字段声明的类型中包含了'BLOB'(无论大小写)，或者没有为该字段声明类型，该字段被指派为NONE亲和性。

注意：如果没有为字段声明类型，该字段的亲和性为NONE，在这种情况下，所有的值都将以它们本身的(或从它们的表示法中推断的)存储类存储。如果你暂时还不确定要往一个字段里放什么内容，或准备将来修改，用NONE亲和性是一个好的选择。但SQLite默认的亲和性是NUMERIC。例如，如果为一定字段声明了类型JUJYFRUIT，该字段的亲和性不是NONE，因为SQLite不认识这种类型，会给它指派默认的NUMERIC亲和性。所以，与其用一个不认识的类型最终得到NUMERIC亲和性，还不如不为它指定类型，从而使它得到NONE亲和性。

亲和性和存储

亲和性对值如何存储到字段有影响，规则如下：

- 一个NUMERIC字段可能包括所有5种存储类。一个NUMERIC字段具有数字存储类的偏好(INTEGER和REAL)。当一个TEXT值被插入到一个NUMERIC字段，将会试图将其转化为INTEGER存储类；如果转化失败，将会试图将其转化为REAL存储类；如果还是失败，将会用TEXT存储类来存储。
- 一个INTEGER字段的处理很像NUMERIC字段。一个INTEGER字段会将REAL值按REAL存储类存储。也就是说，如果这个REAL值没有小数部分，就会被转化为INTEGER存储类。INTEGER字段将会试着将TEXT值按REAL存储；如果转化失败，将会试图将其转化为INTEGER存储类；如果还是失败，将会用TEXT存储类来存储。
- 一个TEXT字段将会把所有的INTEGER或REAL值转化为TEXT。
- 一个NONE字段不试图做任何类型转化。所有值按它们本身的存储类存储。
- 没有字段试图向NULL或BLOB值转化——如无论用什么亲和性。NULL和BLOB值永远都按本来的方式存储在所有字段。

这些规则初看起来比较复杂，但总的设计目标很简单，就是：如果你需要，SQLite会尽量模仿其它的关系型数据库。也就是说，如果你将SQLite看成是一个传统数据库，类型亲和性将会按你的期望来存储值。如果你声明了一个INTEGER字段，并向里面放一个整数，就会按整数来存储。如果你声明了一个具有TEXT, CHAR或VARCHAR类型的字段并向里放一个整数，整数将会转化为TEXT。可是，如果你不遵守这些规定，SQLite也会找到办法来存储你的值。

亲和性的运行

让我们看一些例子来了解亲和性是如何工作的：

```
sqlite> CREATE TABLE domain(i int, n numeric, t text, b blob);
sqlite> INSERT INTO domain VALUES (3.142,3.142,3.142,3.142);
sqlite> INSERT INTO domain VALUES ('3.142','3.142','3.142','3.142');
sqlite> INSERT INTO domain VALUES (3142,3142,3142,3142);
sqlite> INSERT INTO domain VALUES (x'3142',x'3142',x'3142',x'3142');
sqlite> INSERT INTO domain VALUES (null,null,null,null);
sqlite> SELECT ROWID,typeof(i),typeof(n),typeof(t),typeof(b) FROM domain;
```

返回：

```
rowid      typeof(i)  typeof(n) typeof(t)  typeof(b)
1            real         real         text         real
2            real         real         text         text
3            integer    integer    text         integer
4            blob        blob        blob        blob
5            null         null         null         null
```

下面的SQL说明存储类的排序情况：

```
sqlite> SELECT ROWID, b, typeof(b) FROM domain ORDER BY b;
```

返回：

```
rowid b typeof(b)
5 NULL null
1 3.142 real
3 3142 integer
2 3.142 text
4 1B blob
sqlite> SELECT ROWID, b, typeof(b), b<1000 FROM domain ORDER BY b;
```

返回：

```
rowid b typeof(b) b<1000
NULL null NULL
1 3.142 real 1
3 3142 integer 1
2 3.142 text 0
4 1B blob 0
```

存储类和类型转换

关于存储类，需要关注的另一件事是：存储类有时会影响到值如何进行比较。特别是SQLite有时在进行比较之前，会将值在数字存储类(INTEGER和REAL)和TEXT之间进行转换。为进行二进制的比较，遵循如下规则：

- 当一个字段值与一个表达式的结果进行比较，字段的亲和性会在比较之前应用于表达式的结果。
- 当两个字段值进行比较，如果一个字段拥有INTEGER或NUMERIC亲和性而另一个没有，NUMERIC亲和性会应用于非NUMERIC字段的TEXT值。
- 当两个表达式进行比较，SQLite不做任何转换。如果两个表达式有相似的存储类，则直接按它们的值进行比较；否则按类值进行比较。

请看下面例子：

```
sqlite> select ROWID,b,typeof(i),i>'2.9' from domain ORDER BY b;
rowid b typeof(i i>'2.9'
5 NULL null NULL
1 3.142 real 1
3 3142 integer 1
2 3.142 real 1
4 1B blob 1
```

也算是“强类型(STRICT TYPING)”

如果你需要比类型亲和性更强的域完整性，可以使用CHECK约束。你可以使用一个单独的内置函数和一个CHECK约束来实现一个“假的”强类型。

事务

事务定义了一组SQL命令的边界，这组命令或者作为一个整体被全部执行，或者都不执行。事务的典型实例是转帐。

事务的范围

事务由3个命令控制：BEGIN、COMMIT和ROLLBACK。BEGIN开始一个事务，之后的所有操作都可以取消。COMMIT使BEGIN后的所有命令得到确认；而ROLLBACK还原BEGIN之后的所有操作。如：

```
sqlite> BEGIN;
sqlite> DELETE FROM foods;
sqlite> ROLLBACK;
sqlite> SELECT COUNT(*) FROM foods;
COUNT(*)
412
```

上面开始了一个事务，先删除了foods表的所有行，但是又用ROLLBACK进行了回卷。再执行SELECT时发现表中没发生任何改变。

SQLite默认情况下，每条SQL语句自成事务(自动提交模式)。

冲突解决

如前所述，违反约束会导致事务的非法结束。大多数数据库(管理系统)都是简单地将前面所做的修改全部取消。

SQLite有其独特的方法来处理约束违反(或说从约束违反中恢复)，被称为冲突解决。

如：

```
sqlite> UPDATE foods SET id=800-id;
SQL error: PRIMARY KEY must be unique
```

SQLite提供5种冲突解决方案：REPLACE、IGNORE、FAIL、ABORT和ROLLBACK。

- REPLACE: 当发违反了唯一完整性，SQLite将造成这种违反的记录删除，替代以新插入或修改的新记录，SQL继续执行，不报错。
- IGNORE
- FAIL
- ABORT

- ROLLBACK

数据库锁

在SQLite中，锁和事务是紧密联系的。为了有效地使用事务，需要了解一些关于如何加锁的知识。

SQLite采用粗放型的锁。当一个连接要写数据库，所有其它的连接被锁住，直到写连接结束了它的事务。SQLite有一个加锁表，来帮助不同的写数据库都能够在最后一刻再加锁，以保证最大的并发性。

SQLite使用锁逐步上升机制，为了写数据库，连接需要逐级地获得排它锁。SQLite有5个不同的锁状态：未加锁(UNLOCKED)、共享(SHARED)、保留(RESERVED)、未决(PENDING)和排它(EXCLUSIVE)。每个数据库连接在同一时刻只能处于其中一个状态。每种状态(未加锁状态除外)都有一种锁与之对应。

最初的状态是未加锁状态，在此状态下，连接还没有存取数据库。当连接到了一个数据库，甚至已经用BEGIN开始了一个事务时，连接都还处于未加锁状态。

未加锁状态的下一个状态是共享状态。为了能够从数据库中读(不写)数据，连接必须首先进入共享状态，也就是说首先要获得一个共享锁。多个连接可以同时获得并保持共享锁，也就是说多个连接可以同时从同一个数据库中读数据。但哪怕只有一个共享锁还没有释放，也不允许任何连接写数据库。

如果一个连接想要写数据库，它必须首先获得一个保留锁。一个数据库上同时只能有一个保留锁。保留锁可以与共享锁共存，保留锁是写数据库的第1阶段。保留锁即不阻止其它拥有共享锁的连接继续读数据库，也不阻止其它连接获得新的共享锁。

一旦一个连接获得了保留锁，它就可以开始处理数据库修改操作了，尽管这些修改只能在缓冲区中进行，而不是实际地写到磁盘。对读出内容所做的修改保存在内存缓冲区中。

当连接想要提交修改(或事务)时，需要将保留锁提升为排它锁。为了得到排它锁，还必须首先将保留锁提升为未决锁。获得未决锁之后，其它连接就不能再获得新的共享锁了，但已经拥有共享锁的连接仍然可以继续正常读数据库。此时，拥有未决锁的连接等待其它拥有共享锁的连接完成工作并释放其共享锁。

一旦所有其它共享锁都被释放，拥有未决锁的连接就可以将其锁提升至排它锁，此时就可以自由地对数据库进行修改了。所有以前对缓冲区所做的修改都会被写到数据库文件。

死锁

为什么需要了解锁的机制呢？为了避免死锁。

考虑下面表4-7所假设的情况。两个连接——A和B——同时但完全独立地工作于同一个数据库。A执行第1条命令，B执行第2、3条，等等。

表4-7 一个死锁的假设情况

A连接	B连接
sqlite> BEGIN;	
	sqlite> BEGIN;
	sqlite> INSERT INTO foo VALUES ('x');
sqlite> SELECT * FROM foo;	
	sqlite> COMMIT;
	SQL error: database is locked
sqlite> INSERT INTO foo VALUES ('x');	
SQL error: database is locked	

两个连接都在死锁中结束。B首先尝试写数据库，也就拥有了一个未决锁。A再试图写，但当其INSERT语句试图将共享锁提升为保留锁时失败。

为了讨论的方便，假设连接A和B都一直等待数据库可写。那么此时，其它的连接甚至都不能够再读数据库了，因为B拥有未决锁(它能阻止其它连接获得共享锁)。那么时此，不仅A和B不能工作，其它所有进程都不能再操作此数据库了。

如果避免此情况呢？当然不能让A和B通过谈判解决，因为它们甚至不知道彼此的存在。答案是采用正确的事务类型来完成工作。

事务的种类

SQLite有三种不同的事务，使用不同的锁状态。事务可以开始于：DEFERRED、MMEDIATE或EXCLUSIVE。事务类型在BEGIN命令中指定：

```
BEGIN [ DEFERRED | IMMEDIATE | EXCLUSIVE ] TRANSACTION;
```

一个DEFERRED事务不获取任何锁(直到它需要锁的时候)，BEGIN语句本身也不会做什么事情——它开始于UNLOCK状态。默认情况下就是这样的，如果仅仅用BEGIN开始一个事务，那么事务就是DEFERRED的，同时它不会获取任何锁；当对数据库进行第一次读操作时，它会获取SHARED锁；同样，当进行第一次写操作时，它会获取RESERVED锁。

由BEGIN开始的IMMEDIATE事务会尝试获取RESERVED锁。如果成功，BEGIN IMMEDIATE保证没有别的连接可以写数据库。但是，别的连接可以对数据库进行读操作；但是，RESERVED锁会阻止其它连接的BEGIN IMMEDIATE或者BEGIN EXCLUSIVE命令，当

其它连接执行上述命令时，会返回SQLITE_BUSY错误。这时你就可以对数据库进行修改操作了，但是你还不能提交，当你COMMIT时，会返回SQLITE_BUSY错误，这意味着还有其它的读事务没有完成，得等它们执行完后才能提交事务。

EXCLUSIVE事务会试着获取对数据库的EXCLUSIVE锁。这与IMMEDIATE类似，但是一旦成功，EXCLUSIVE事务保证没有其它的连接，所以就可对数据库进行读写操作了。

上节那个例子的问题在于两个连接最终都想写数据库，但是它们都没有放弃各自原来的锁，最终，SHARED锁导致了问题的出现。如果两个连接都以BEGIN IMMEDIATE开始事务，那么死锁就不会发生。在这种情况下，在同一时刻只能有一个连接进入BEGIN IMMEDIATE，其它的连接就得等待。BEGIN IMMEDIATE和BEGIN EXCLUSIVE通常被写事务使用。就像同步机制一样，它防止了死锁的产生。

基本的准则是：如果你正在使用的数据库没有其它的连接，用BEGIN就足够了。但是，如果你使用的数据库有其它的连接也会对数据库进行写操作，就得使用BEGIN IMMEDIATE或BEGIN EXCLUSIVE开始你的事务。

数据库管理

数据库管理用于控制数据库如何操作。从SQL的角度，数据库管理包括一些主题如会视图(view)、触发器(trigger)和索引(indexe)。另外，SQLite包括自己一些独特的管理，如数据库pragma，可以用来配置数据库参数。

视图

物化的视图

在关系模型中称为数据可修改的视图。

索引

索引的利用

理解索引何时被利用及何时不被利用是重要的。SQLite有明确的条件来决定是否使用索引。如果可能，在WHERE子名中有下列表达式时，SQLite将使用单字段索引：

```
column {=|>|>=|<=|<} expression
expression {=|>|>=|<=|<} column
column IN (expression-list)
column IN (subquery)
```

多字段索引的使用有很明确的条件。这最好用例子来说。假设你有如下定义的一个表：

```
CREATE TABLE foo (a,b,c,d);
```

触发器

当特定的表上发生特定的数据库事件时，触发器会执行特定的SQL命令。创建触发器的一般语法如下：

```
CREATE [TEMP|TEMPORARY] TRIGGER name [BEFORE|AFTER]
 [INSERT|DELETE|UPDATE|UPDATE OF columns] ON table
action
```

UPDATE触发器

不同于INSERT and DELETE触发器，UPDATE触发器可以定义在一个表的特定的字段上。The general form of this kind of trigger is as follows:

```
CREATE TRIGGER name [BEFORE|AFTER] UPDATE OF column ON table
action
```

The following is a SQL script that shows an UPDATE trigger in action:

```
.h on
.m col
.w 50
.echo on
CREATE TEMP TABLE log(x);
CREATE TEMP TRIGGER foods_update_log UPDATE of name ON foods
BEGIN
    INSERT INTO log VALUES('updated foods: new name=' || NEW.name);
END;
BEGIN;
UPDATE foods set name='JUJYFRUIT' where name='JujyFruit';
SELECT * FROM log;
ROLLBACK;
```

错误处理

定义一个事件的before触发器给了你一个阻止事件发生的机会。before触发器可以实现新的完整性约束。SQLite为触发器提供了一个称为RAISE()的特殊SQL函数，可以在触发器体中唤起一个错误。RAISE如下定义：

```
RAISE(resolution, error_message);
```

使用触发器的外键约束

在SQLite中，触发器最有趣的应用之一是实现外键约束。为了进一步了解触发器，我将用这个想法在foods表和food_types表之间实现外键。

附加(Attaching)数据库

SQLite允许你用ATTACH命令将多个数据库“附加”到当前连接上来。当你附加了一个数据库，它的所有内容在当前数据库文件的全局范围内都是可存取的。ATTACH的语法为：

```
ATTACH [DATABASE] filename AS database_name;
```

清洁数据库

SQLite有两个命令用于数据库清洁——REINDEX和VACUUM。REINDEX用于重建索引，有两种形式：

```
REINDEX collation_name;
REINDEX table_name|index_name;
```

第一种形式利用给定的排序法名称重新建立所有的索引。

VACUUM通过重建数据库文件来清除数据库内所有的未用空间。

数据库配置

SQLite没有配置文件。所有这些配置参数都是用pragma来实现。Pragma以独特的方式工作，有些像变量，又有些像命令。

连接缓冲区大小

缓冲区尺寸pragma控制一个连接可以在内存中使用多少个数据库页。要查看当前缓冲区大小的默认值，执行：

```
sqlite> PRAGMA cache_size;
cache_size
2000
```

要改变缓冲区大小，执行：

```
sqlite> PRAGMA cache_size=10000;
sqlite> PRAGMA cache_size;
cache_size
10000
```

获得数据库信息

可以使用数据库的schema pragma来获得数据库信息，定义如下：

- database_list: Lists information about all attached databases.
- index_info: Lists information about the columns within an index. It takes an index name as an argument.
- index_list: Lists information about the indexes in a table. It takes a table name as an argument.
- table_info: Lists information about all columns in a table.

请看下面示例：

```
sqlite> PRAGMA database_list;
seq name file
0 main /tmp/foods.db
2 db2 /tmp/db
sqlite> CREATE INDEX foods_name_type_idx ON foods(name,type_id);
sqlite> PRAGMA index_info(foods_name_type_idx);
seqn cid name
0 2 name
1 1 type_id
sqlite> PRAGMA index_list(foods);
seq name unique
0 foods_name_type_idx 0
sqlite> PRAGMA table_info(foods);
cid name type notn dflt pk
0 id integer 0 1
1 type_id integer 0 0
2 name text 0 0
```

页大小、编码和自动排空

The database page size, encoding, and autovacuuming must be set before a database is created. That is, in order to alter the defaults, you must first set these pragmas before creating any database objects in a new database. The defaults are a 1,024-byte page size and UTF-8 encoding. SQLite supports page sizes ranging from 512 to 32,786 bytes, in powers of 2. Supported encodings are UTF-8, UTF-16le (little-endian UTF-16 encoding), and UTF-16be (big-endian UTF-16 encoding).

如果使用auto_vacuum pragma，可以使数据库自动维持最小。一般情况下，当一个事务从数据库中删除了数据并提交后，数据库文件的大小保持不变。当使用了auto_vacuum pragma后，当删除事务提交时，数据库文件会自动缩小。

系统表

sqlite_master表是一个系统表，它包含数据库中所有表、视图、索引和触发器的信息。例如，foods的当前内容如下：

```
sqlite> SELECT type, name, rootpage FROM sqlite_master;
type        name              rootpage
table              episodes 2
table              foods 3
table              foods_episodes 4
table              food_types 5
index      foods_name_idx 30
table              sqlite_sequence 50
trigger     foods_update_trg 0
trigger     foods_insert_trg 0
trigger     foods_delete_trg 0
```

有关sqlite_master表的结构请参考第2章的“获得数据库的Schema信息”一节。

sqlite_master包含一个称为sql的字段，存储了创建对象的DDL命令，如：

```
sqlite> SELECT sql FROM sqlite_master WHERE name='foods_update_trg';
```

返回：

```
CREATE TRIGGER foods_update_trg
BEFORE UPDATE OF type_id ON foods
BEGIN
 SELECT CASE
 WHEN (SELECT id FROM food_types WHERE id=NEW.type_id) IS NULL
 THEN RAISE( ABORT,
 'Foreign Key Violation: foods.type_id is not in food_types.id')
END;
END
```

查看**Query**的执行

可以用EXPLAIN命令查看SQLite执行一个查询的方法。EXPLAIN列出一个SQL命令编译后的VDBE程序。

```
sqlite> .m col
sqlite> .h on
sqlite> .w 4 15 3 3 3 10 3
sqlite> EXPLAIN SELECT * FROM foods;
addr  opcode           p1   p2   p3   p4          p5   comment
----  ---------------  ---  ---  ---  ----------  ---  -------
0     Trace            0    0    0                00
1     Goto             0    11   0                00
2     OpenRead         0    7    0    3           00
3     Rewind           0    9    0                00
4     Rowid            0    1    0                00
5     Column           0    1    2                00
6     Column           0    2    3                00
7     ResultRow        1    3    0                00
8     Next             0    4    0                01
9     Close            0    0    0                00
10    Halt             0    0    0                00
11    Transaction      0    0    0                00
12    VerifyCookie     0    40   0                00
13    TableLock        0    7    0    foods       00
14    Goto             0    2    0                00
```

第5章 设计和概念

本章为后面的3章打下基础，这几章专注于SQLite编程。这几章专注于作为程序员，在编码时你所应该了解的有关SQLite的东西。无论你是用C语言对SQLite进行编程，还是用其它的编程语言，这些内容都是重要的。它不仅帮助你了解API，还包括部分有关SQLite的体系结构和实现方法的内容。具备了这些知识，你就可以编出更好的代码，这些代码执行得更快，且不会产生死锁、不可预知错误等问题。你会看到SQLite如何处理你的代码，你还会变得更加自信，因为你知道自己正前进在解决问题的正确方向上。

你不需要从头到尾地读内部代码才能理解这些内容，你也不必是一个C程序员。SQLite的设计和概念都是非常直观和容易理解的。只有一小部分内容你需要知道，本章就介绍这些内容。

1．明显地，你需要知道API是如何工作的。于是本章从一个对API概念性的介绍开始，图示了主要的数据结构，API的一般设计和它主要的函数。还可以看到SQLite的一些主要的子系统，这些子系统在查询的处理过程中起着重要作用。

2．除了知道什么函数做什么，你还需要从比API高的角度来看问题，看看所有这些函数在事务(transactions)中是如何操作的。SQLite的所有的工作都是在事务中完成的。于是，你需要知道在API之下，事务如何按照锁的约束来工作。如果你不知道锁是如何操作的，这些锁就会导致问题。当对锁有所了解之后，你不仅可以避免潜在的并发问题，还可以通过编程控制它们来优化你的查询。

3．最后，你还必须理解如何将这些内容应用于编码。本章的最后部分会将3个主题结合在一起——API、事务和锁，并且看一看好代码与坏代码的区别。

空注：本章看得还是比较仔细的，翻译的也比较全。

API

从功能的角度来区分，SQLite的API可分为两类：核心API的扩充API。核心API由所有完成基本数据库操作的函数构成，包括：连接数据库、执行SQL和遍历结果集。它还包括一些功能函数，用来完成字符串格式化、操作控制、调试和错误处理等任务。扩充API提供不同的方法来扩展SQLite，它使你能够创建自定义的SQL扩展，并与SQLite本身的SQL相集成。

SQLite版本3的新特性

在开始之前，我们先讨论一下SQLite版本3的新特色：

一、首先，SQLite的API被彻底重新设计了，并具有了许多新特性。由第二版的15个函数增加到88个函数。这些函数包括支持UTF-8和UTF-16编码的功能函数。SQLite3有一个更方便的查询模式，使查询的预处理更容易并且支持新的参数绑定方法。SQLite3还增加了用户定义的排序序列、CHECK约束、64位的键值和新的查询优化。

二、在后端大大地改进了并发性能。加锁子系统引进了一种新的锁升级模型，解决了第二版中的写进程饿死的问题。这种模型保证写进程按照先来先服务的算法得到排它锁(Exclusive Lock)。甚至，写进程通过把结果写入临时缓冲区(Temporary Buffer)，可以在得到排它锁之前就开始工作。这对于写要求较高的应用，性能可提高400%。

三、SQLite 3包含一个改进了的B-tree模型。现在对库表使用B+tree，大大提高查询效率，存储大数据字段更有效，并可以从磁盘上删除不用了的字段。其结果是数据库文件的体积减小了25–35%并改善了全面性能。B+tree将在第9章介绍。

四、SQLite 3最重要的改变是它的存储模型。由第二版只支持文本模型，扩展到支持5种本地数据类型，如第4章所介绍的，还增强了弱类型和类型亲和性的概念。每种类型都被优化，以得到更高的查询性能并战用更少的存储空间。例如，整数和浮点数以二进制的形式存储，而不再是以ASCII形式存储，这样，就不必再对WHERE子句中的值进行转换(像第2版那样)。弱类型使你能够在定义一个字段时选择是否预声明类型。亲和性确定一个值存储于字段的格式——基于值的表示法和列的亲和性。类型亲和性与弱类型紧密关联——列的亲和性由其类型的声明确定。

在很多方面，SQLite 3是一个与SQLite 2完全不同的数据库，并且提供了很多在适应性、特色和性能方面的改进。

主要的数据结构

在第1章你看到了很多SQLite组件——分词器、分析器和虚拟机等等。但是从程序员的角度，最需要知道的是：connection、statements、B-tree和pager。它们之间的关系如图5-1所示。这些对象构成了编写优秀代码所必须知道的3个首要内容：API、事务和锁。

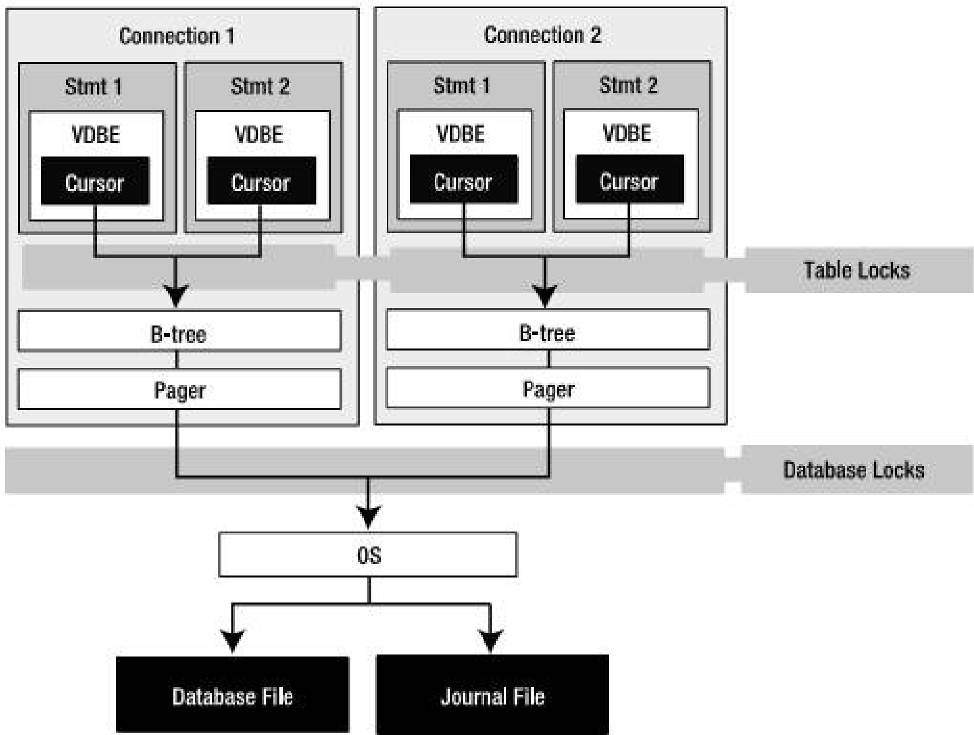


图5-1 SQLite C API对象模型

从技术上来说，B-tree和pager不是API的一部分，但是它们却在事务和锁上起着关键作用。这里只介绍关联的内容，详细内容将在“事务”一节介绍。

连接(Connection)和语句(Statement)

连接(Connection)和语句(Statement)是执行SQL命令涉及的两个主要数据结构，几乎所有通过API进行的操作都要用到它们。连接代表在一个独立的事务环境下的一个单独的数据库连接。每个语句都和一个连接关联，通常表示一个编译过的SQL语句。在内部，它以VDBE字节码表示。语句包括执行一个命令所需要一切，包括保存VDBE程序执行状态所需的资源，指向硬盘记录的B-tree游标，以及参数等等。

B-tree和Pager

一个连接可以有多个database对象——一个主数据库和附加的数据库。每一个数据库对象有一个B-tree对象，一个B-tree有一个pager对象(这里的对象不是面向对象的“对象”，只是为了说清楚问题)。

语句最终都是通过连接的B-tree和pager从数据库读或者写数据，通过B-tree的游标(cursor)遍历存储在页(page)中的记录。在游标访问页之前，页必须从磁盘加载到内存，而这就是pager的任务。任何时候，如果B-tree需要页，它都会请求pager从磁盘读取数据，pager把页加载到页缓冲区(page cache)。之后，B-tree和与之关联的游标就可以访问位于页中的记录了。

如果游标改变了页，为了防止事务回滚，pager必须采取特殊的方式保存原来的页。总的来说，pager负责读写数据库，管理内存缓存和页，以及管理事务、锁和崩溃恢复(这些在“事务”一节会详细介绍)。

总之，关于连接和事务，你必须知道两件事：(1)对数据库的任何操作，一个连接存在于一个事务之下。(2)一个连接绝不会同时存在于多个事务之下。无论何时，一个连接在对数据库做任何操作时，都总是在恰好一个事务之下，不会多，也不会少。

核心API

核心API主要与执行SQL命令有关。有两种方法执行SQL语句：预编译查询和封装查询。预编译查询由三个阶段构成：准备(preparation)、执行(execution)和定案(finalization)。其实封闭装查询只是对预编译查询的三个过程进行了包装而已，最终也会转化为预编译查询来执行。

连接的生命周期(The Connection Lifecycle)

和大多数据库连接相同，其生命周期由三个阶段构成：

1. 连接数据库(Connect to the database)。

2. 处理事务(Perform transactions)：如你所知，任何命令都在事务下执行。默认情况下，事务自动提交，也就是每一个SQL语句都在一个独立的事务下运行。当然也可以通过使用BEGIN..COMMIT手动提交事务。

3. 断开连接(Disconnect from the database)：关闭数据库文件。还要关闭所有附加的数据库文件。

在查询的处理过程中还包括其它一些行为，如处理错误、“忙”句柄和schema改变等，所有这些都将在utility functions一节中介绍。

连接数据库(Connect to the database)：

连接数据库不只是打开一个文件。每个SQLite数据库都存储在单独的操作系统文件中——数据库与文件一一对应。连接、打开数据库的C API为sqlite3_open()，它只是一个简单的系统调用，来打开一个文件，它的实现位于main.c文件中。

SQLite还可以创建内存数据库。如果你使用:memory:或一个空字符串做数据库名，数据库将在RAM中创建。内存数据库将只能被创建它的连接所存取，不能与其它连接共享。另外，内存数据库只能存活于连接期间，一旦连接关闭，数据库就将从内存中被删除。

当连接一个位于磁盘上的数据库时，如果数据库文件存在，则打开该文件；如果不存在，SQLite会假定你想创建一个新的数据库。在这种情况下，SQLite不会立即在磁盘上创建一个文件，只有当你向数据库写入数据时才会创建文件，比如：创建表、视图或者其它数据库对

象。如果你打开一个数据库，不做任何事，然后关闭它，SQLite会创建一个文件，但只是一个长度为0的空文件而已。

另外一个不立即创建新文件的原因是，一些数据库的参数，比如：编码，页大小等，只能在数据库创建之前设置。默认情况下，页大小为1024字节，但是你可以选择512-32768字节之间为 2幂数的数字。有些时候，较大的页能更有效地处理大量的数据。你可以使用page_size pragma来设置数据库页大小。

字符编码是数据库的另一个永久设置。你可以使用encoding pragma来设置字符编码，其值可以是UTF-8、UTF-16、UTF-16le (little endian)和UTF-16be (big endian)。

执行预处理查询

前面提到，预处理查询(Prepared Query)是SQLite执行所有SQL命令的方式，包括以下三个步聚：

(1) 准备(preparation)：

分词器(tokenizer) 、分析器(parser)和代码生成器(code generator)把SQL语句编译成VDBE字节码，编译器会创建一个语句句柄(sqlite3_stmt)，它包括字节码以及其它执行命令和遍历结果集所需的全部资源。相应的C API为sqlite3_prepare()，位于prepare.c文件中。

(2) 执行(execution)：

虚拟机执行字节码，执行过程是一个步进(stepwise)的过程，每一步(step)由sqlite3_step()启动，并由VDBE执行一段字节码。当第一次调用sqlite3_step()时，一般会获得一种锁，锁的种类由命令要做什么(读或写)决定。对于SELECT语句，每次调用sqlite3_step()使用语句句柄的游标移到结果集的下一行。对于结果集中的每一行，它返回SQLITE_ROW，当到达结果末尾时，返回SQLITE_DONE。对于其它SQL语句(INSERT、UPDATE、DELETE等)，第一次调用sqlite3_step()就导致VDBE执行整个命令。

(3) 定案(finalization)：

VDBE关闭语句，释放资源。相应的C API为sqlite3_finalize()，它导致VDBE结束程序运行并关闭语句句柄。如果事务是由人工控制开始的，它必须由人工控制进行提交或回卷，否则sqlite3_finalize()会返回一个错误。当sqlite3_finalize()执行成功，所有与语句对象关联的资源都将被释放。在自动提交模式下，还会释放关联的数据库锁。

每一步(preparation、execution和finalization)都关联于语句句柄的一种状态(prepared、active和finalized)。Pepared表示所有资源都已分配，语句已经可以执行，但还没有执行。现在还没有申请锁，一直到调用sqlite3_step()时才会申请锁。Active状态开始于对sqlite3_step()的调用，此时语句正在被执行并拥有某种锁。Finalized意味着语句已经被关闭且所有相关资源已经被释放。通过图5-2可以更容易地理解该过程：

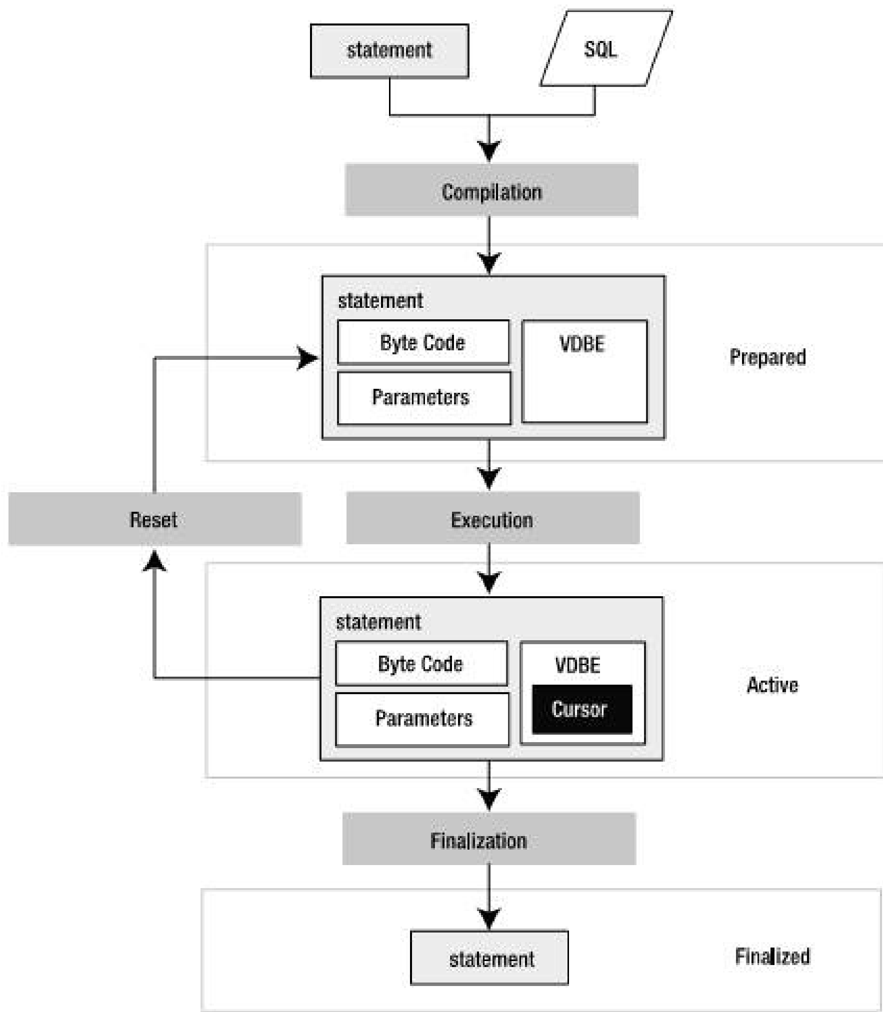


图5-2 语句处理

下面代码例示了在SQLite上执行一个query的一般过程。

```
#include<stdio.h>
#include<stdlib.h>
#include"sqlite3.h"
#include<string.h>
#pragma comment(lib, "sqlite3.lib")
int main(int argc,char **argv)
{
        int rc,i,ncols;
        sqlite3 *db;
        sqlite3_stmt *stmt;
        char *sql;
        const char*tail;
        //打开数据
        rc=sqlite3_open("foods.db",&db);
        if(rc){
                fprintf(stderr,"Can'topendatabase:%sn",sqlite3_errmsg(db));
                sqlite3_close(db);
                exit(1);
        }
        sql="select * from episodes";
        //预处理
        rc=sqlite3_prepare(db,sql,(int)strlen(sql),&stmt,&tail);
        if(rc!=SQLITE_OK){
                fprintf(stderr,"SQLerror:%sn",sqlite3_errmsg(db));
        }
        rc=sqlite3_step(stmt);
        ncols=sqlite3_column_count(stmt);
        while(rc==SQLITE_ROW){
                for(i=0;i<ncols;i++){
                        fprintf(stderr,"'%s'",sqlite3_column_text(stmt,i));
                }
                fprintf(stderr,"\n");
                rc=sqlite3_step(stmt);
        }
        //释放statement
        sqlite3_finalize(stmt);
        //关闭数据库
        sqlite3_close(db);
        printf("\n");
        return(0);
}
```

空注：

上述代码在VC6++下调试通过，其步骤为：

将上述代码做成一个.cpp文件并为它创建工作空间。

将sqlite3.def和sqlite3.dll文件复制到工作空间所在目录。(这两个文件可由sqlitedll-3_6_18.zip文件解压而得)

进入DOS命令行状态，进入工作空间所在目录，执行如下3条命令：

```
PATH = D:\Program Files\Microsoft Visual Studio 9.0\VC\bin;%PATH%
PATH = D:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE;%PATH%
LIB /DEF:sqlite3.def /machine:IX86
```

注：上述命令用于制作.lib文件，用于项目的链接。如果PATH已经设好，前两条命令可能不需要执行；如果执行，可能需要根据VC的安装情况有所改动。

临时存储器：

临时存储器是查询处理的重要组成部分。SQLite有时需要存储命令执行过程中产生的中间结果——如结果集由ORDER BY子句指定需要排序，或进行多表交叉查询时。中间结果存储在临时存储器中。临时存储器或者在内存，或者在文件中。

使用参数化的SQL

SQL语句可以包含参数。参数是SQL命令中的占位符，它们的值会在编译之后提供(称为“绑定”)。下面是带参数的SQL示例语句：

```
INSERT INTO foods (id, name) VALUES (?,?);
INSERT INTO episodes (id, name) (:id, :name);
```

上述语句表现了参数的两种绑定方式：按位置和按名称。第1条命令采用了位置参数，第2条命令采用了名称参数。

Positional parameters are defined by the position of the question mark in the statement. The first question mark has position 1, the second 2, and so on. Named parameters use actual variable names, which are prefixed with a colon. When sqlite3_prepare() compiles a statement with parameters, it allocates placeholders for the parameters in the resulting statement handle. It then expects values to be provided for these parameters before the statement is

executed. 如果你没有为参数绑定值，在语句执行时，SQLite 默认为各参数赋NULL值。

使用参数绑定的好处是你可以多次执行相同的语句而不必重新编译它们。You just reset the statement, bind a new set of values, and reexecute. This is where resetting rather than finalizing a statement comes in handy: it avoids the overhead of SQL compilation. By resetting a statement, you are reusing the compiled SQL code. You completely avoid the tokenizing, parsing, and code generation overhead. Resetting a statement is implemented in the API by the sqlite3_reset() function.

使用参数的另一个好处是：SQLite可以对你绑定到参数的字符串值做一定的处理。例如，有一个参数值'Kenny's Chicken'，参数绑定过程会自动地将其转化为'Kenny''s Chicken'。下面的伪代码说明了绑定参数的基本方法：

```
db = open('foods.db')
stmt = db.prepare('INSERT INTO episodes (id, name) VALUES (:id, :name)')
stmt.bind('id', '1')
stmt.bind('name', 'Soup Nazi')
stmt.step()
# Reset and use again
stmt.reset()
stmt.bind('id', '2')
stmt.bind('name', 'The Junior Mint')
# Done
stmt.finalize()
db.close()
```

执行封装的Query

如前文所述，有两个很有用的函数可以封装查询的预编译过程，允许你在单一的函数调用中执行SQL命令。一个函数是sqlite3_exec()，特别适合执行不需要返回数据的查询。另一个是sqlite3_get_table()，返回一个表格化的结果集。In many language extensions you will see analogs to both functions. Most extensions refer to the first method simply as exec(), and the second as just get_table().

sqlite3_get_table()的函数名起得不太合适，听起来好象是要返回一个表的全部数据。其实它的命名只是表示将会返回一个表格化的结果集。

```
db = open('foods.db')
table = db.get_table("SELECT * FROM episodes LIMIT 10")
for i=0; i < table.rows; i++
  for j=0; j < table.cols; j++
    print table[i][j]
  end
end
db.close()
```

错误处理

前面的例子都是极度简化了的，只关注查询的执行。而在实际情况下，你总得关注出错的可能性。你前面所看到的几乎每个函数在某些情况下都可能引发错误。通常你需要对错误代码SQLITE_ERROR、SQLITE_BUSY和SQLITE_SCHEMA进行处理。SQLITE_BUSY在当前连接不能够获得一个锁时触发，SQLITE_SCHEMA在语句的编译与执行之间schema发生了改变时触发。“忙”状态将在本章的事务一节中介绍。Schema错误将在第6章介绍。

很多语言扩展难于处理schema错误。有些透明地报告处于忙状态，有些直接返回实际的错误代码。无论如何，如果你遇到了schema错误，表示有其它的连接在你的读与写之间改变了数据库，你的语句已不再合法。你需要重新编译语句，以便能够重新执行它。Schema错误只会发生在对prepare()的调用和第1次对step()的调用之间。如果你的第1次step()调用成功，那你就不必再担心在后面调用step()时会引发schema错误了，因为你的连接已经锁住了数据库，其它的连接不可能在此期间修改数据库。

对于一般性错误，API提供了sqlite3_errcode()来获取最后一次调用API函数时的返回码。你可以使用sqlite3_errmsg()函数得到更具体的错误信息，该函数提供了对最后错误的文字描述，大多数语言扩展都支持这个函数。

有了这个观念，前面例子中的每个调用都可以用类似下面的代码来检查错误：

```
# Check and report errors
if db.errcode() != SQLITE_OK
  print db.errmsg(stmt)
end
```

一般情况下，错误处理并不困难。处理错误的方法由你确切地想要做什么决定。

格式化SQL语句

另一个方便的函数是sqlite3_mprintf()，它是标准C库函数sprintf()的一个变体。它有很独特的替换符，特别方便对SQL进行处理。它的替换符是%q和%Q。%q的工作原理像%s，从参数列表中取得一个以NULL结束的字符串。它会将单引号反斜杠都双写，使你更容易防范SQL注入式攻击(参本节下文的“SQL注入式攻击”一段)。例如：

```
char* before = "Hey, at least %q no pig-man.";
char* after = sqlite3_mprintf(before, "\he's\");
```

上述程序执行后after的值为'Hey, at least \he''s\ no pig-man'。The single quote in he’s is doubled along with the backslashes around it, making it acceptable as a string literal in a SQL statement. The %Q formatting does everything %qdoes, but it additionally encloses the resulting string in single quotes. Furthermore, if the argument for %Q is a NULL pointer (in C), it produces the string NULL without single quotes. For more information, see the sqlite3_mprintf() documentation in the C API reference in Appendix B.

SQL注入式攻击：

如果你的应用程序依赖用户的输入来构造SQL语句，那么它将很容易受到SQL注入攻击。如果你没有精心地过滤用户输入，有人可能会输入别有用心的内容，注入到你的SQL中，并在其后面构成一个新的SQL语句。例如，你的程序用用户输入来填充下面SQL语句：

```
SELECT * FROM foods WHERE name='%s';
```

如果无论用户输入什么都直接来替换%s，如果用户对你的数据库有一定了解，他可以输入如下内容：

```
nothing' LIMIT 0; SELECT name FROM sqlite_master WHERE name='%
```

将用户输入替换进原有的SQL语句之后，变成了两个新的语句：

```
SELECT * FROM foods WHERE name='nothing' LIMIT 0; SELECT name FROM
sqlite_master WHERE name='%';
```

第1个语句什么都不返回，但第2个将返回表中所有的记录。Granted, the odds of this happening require quite a bit of knowledge on the attacker’s part, but it is nevertheless possible. Some major (commercial) web applications have been known to keep SQL statements embedded in their JavaScript, which can provide plenty of hints about the database being used. In the previous example, all a malicious user has to do now is insert DROP TABLE statements for every table found in sqlite_master and you could find yourself fumbling through backups.

操作控制

API中包含几个命令来监视、控制，或者说限制数据库操作。SQLite使用过滤(或称回叫)函数来完成此功能，你可以注册它们由特定的事件来调用。有3个“hook”函数：sqlite3_commit_hook()，它监视事务的提交；sqlite3_rollback_hook()，它监视事务的回卷；sqlite3_update_hook()，它监视INSERT、UPDATE和DELETE操作。这些函数在运行时被调用——即当命令执行时被调用。Each hook allows you to register a callback function on a connection-by-connection basis, and lets you provide some kind of application-specific data to be passed to the callback as well. The general use of operational control functions is as follows:

```
def commit_hook(cnx)
       log('Attempted commit on connection %x', cnx)
       return -1
end
db = open('foods.db')
db.set_commit_hook(rollback_hook, cnx)
db.exec("BEGIN; DELETE from episodes; ROLLBACK")
db.close()
```

使用线程

SQLite有几个可以在多线程环境下使用的函数。在3.3.1版中，SQLite引入了一种称为共享缓冲区模式的独特的操作模式，就是为多线程的内嵌式服务器设计的。这个模式提供了一种用单线程来处理多连接的方法，可以共享相同的页缓冲区，从而降低了整个服务器的内存需求。这个模式包括多个函数来管理内存和服务器。详见第6章“共享缓冲区模式”一节。

扩充API

SQLite的扩充API用来支持用户定义的函数、聚合和排序法。用户定义函数是一个SQL函数，它对应于你用C语言或其它语言实现的函数的句柄。使用C API时，这些句柄用C/C++实现。

用户定义的扩展必须注册到一个由连接到连接的基础(connection-by-connection basis)之上，存储在程序内存中。也就是说，它们不是存储在数据库中(就像大型数据库的存储过程一样)，而是存储在你的程序中。

创建用户自定义函数

实现一个用户自定义的函数分为两步。首先，写句柄。句柄实现一些你想通过SQL完成的功能。然后，注册句柄，为它提供SQL名称、参数的数量和一个指向句柄的指针。

例如，你想创建一个叫做hello_newman()的SQL函数，它返回文本'Hello Jerry'。在SQLite C API中，先创建一个C函数来实现此功能，如：

```
void hello_newman(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    /* Create Newman's reply */
    const char *msg = "Hello Jerry";
    /* Set the return value. Have sqlite clean up msg w/ sqlite_free(). */
    sqlite3_result_text(ctx, msg, strlen(msg), sqlite3_free);
}
```

不了解C和C API也没关系。这个句柄仅返回'Hello Jerry'。下面是实际使用它。使用sqlite3_create_function()函数注册这个句柄：

```
    sqlite3_create_function(db, "hello_newman", 0, hello_newman);
```

第1个参数(db)是数据库连接。第2个参数是函数的名称，这个名称将出现在SQL中。第3个参数表示这个函数有0个参数(如果该参数值为-1，表示该函数接受可变数量的参数)。最后一个参数是C函数hello_newman()的指针，当SQL函数被调用时，通过这个指针来调用实际的函数。

一旦进行了注册，SQLite就知道了当遇到SQL函数调用hello_newman()时，它需要调用C函数hello_newman()来得到结果。现在，你可以在程序中执行SELECT hello_newman()语句，它将返回单行单列的文本'Hello Jerry'。

如前所述，很多语言扩展允许用各自的语言来实现用户自定义的函数。例如，Java、Perl等。不同的语言扩展用不同的方法注册函数，有些使用其本身语言的函数来完成此项工作，例如，在Ruby中使用block—one。

创建用户自定义聚合

所谓聚合函数，就是那些在结果集中应用于全部记录，并从中计算一些聚合值的函数。SUM()、COUNT()和AVG()都是SQLite标准聚合函数的例子。

创建用户自定义聚合需要三步：注册聚合、实现步进函数(对结果集中的每条记录调用)、实现定案函数(在所有记录处理完后调用)。在定案函数中计算最终的聚合值，并做一些必要的清理工作。

创建用户自定义排序法

事务

现在，你应该已经对API有了一个较好的了解。你知道了执行SQL命令的不同方法和一些有用的功能函数。但是，只知道API还不够，事务和锁与查询的处理过程是紧密关联的。查询永远只能在事务中完成，事务包含锁，而如果不清楚自己到底在做什么，锁则可能会导致问题。根据你是如何使用SQL及如何编码，你可以控制锁的类型和持续时间。

第4章图示了一个特殊的假想：两个独立的连接同时执行时导致了死锁。作为程序员，你还可以从代码的角度来看待问题，代码中可能包括处于多种状态的多个连接，带有多个语句句柄，而你的代码可能在你不知情的情况下就持有了EXCLUSIVE锁，从而使其它连接不能做任何事情。

这就是为什么掌握下面知识很重要：事务和锁如何工作，它们如何与API结合来处理查询。理想的目标是，你应该能够看着你写的代码并说出事务处于什么状态，或者至少能够发现潜在的问题。本节将介绍事务和锁背后的运行机制，下一节将介绍如何实际地编码。

事务的生命周期

有一些关于代码和事务的问题需要关注。首先需要知道哪个对象运行在哪个事务之下。另一个问题是持续时间——一个事务何时开始，何时结束，从哪一点开始影响其它连接？第一个问题与API直接关联，第二个与SQL的一般概念及SQLite的特殊实现关联。

一个连接(connection)可以包含多个语句(statement)，而且每个连接有一个与数据库关联的B-tree和一个pager。Pager在连接中起着很重要的作用，因为它管理事务、锁、内存缓冲以及负责崩溃恢复(crash recovery)。当你进行数据库写操作时，记住最重要的一件事：在任何时候，只在一个事务下执行一个连接。这回答了第一个问题。

关于第二个问题，一般来说，一个事务的生命周期和语句差不多，你也可以手动结束它。默认情况下，事务自动提交，当然你也可以通过BEGIN..COMMIT手动提交。接下来的问题是事务如何与锁关联。

锁的状态

大多数情况下，锁的生命周期在事务的生命周期之中。它们不一定同时开始，但总时同时结束。当你结束一个事务时，也会释放它相关的锁。或者说，锁直到事务结束或系统崩溃时才会释放。如果系统在事务没有结束的情况下崩溃，那么下一个访问数据库的连接会处理这种情况，详见“锁与崩溃恢复”一节。

在SQLite中有5种不同的锁状态，连接(connection)任何时候都处于其中的一个状态。图5-3显示了锁的状态以及状态的转换。

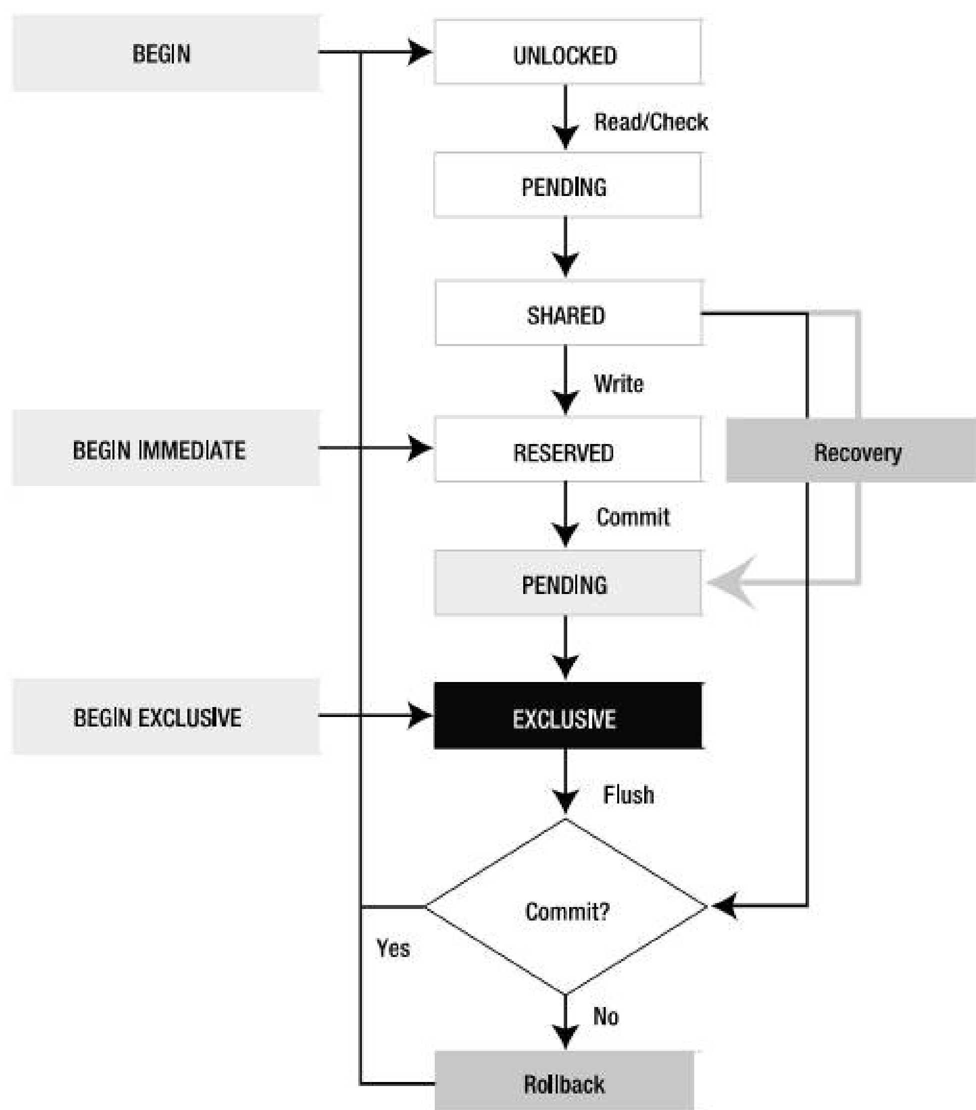


图5-3 SQLite锁转换

关于这个图有以下几点值得注意：

(1)一个事务可以在UNLOCKED、RESERVED或EXCLUSIVE三种状态下开始。默认情况下在UNLOCKED时开始。

(2)白色框中的UNLOCKED、PENDING、SHARED和 RESERVED可以在一个数据库的同一时刻存在。

(3)从灰色的PENDING开始，事情就变得严格起来，意味着事务想得到排它锁(EXCLUSIVE)(注意与白色框中的区别)。

虽然锁有这么多状态，但是从体质上来说，只有两种情况：读事务和写事务。

读事务

我们先来看看SELECT语句执行时锁的状态变化过程，非常简单：一个连接执行SELECT语句，触发一个事务，从UNLOCKED到SHARED，当事务COMMIT时，又回到UNLOCKED，就这么简单。

那么，当你运行两个语句时会发生什么呢？这时如何加锁呢？这依赖于你是否运行在自动提交状态。考虑下面的例子(为了简单，这里用了伪码)：

```
db = open('foods.db')
db.exec('BEGIN')
db.exec('SELECT * FROM episodes')
db.exec('SELECT * FROM episodes')
db.exec('COMMIT')
db.close()
```

由于显式地使用了BEGIN和COMMIT，两个SELECT命令在一个事务下执行。第一个exec()执行时，连接处于SHARED，然后第二个exec()执行。当事务提交时，连接又从SHARED回到UNLOCKED状态，状态变化如下：

```
UNLOCKED→PENDING→SHARED→UNLOCKED
```

如果没有BEGIN和COMMIT两行，两个SELECT都运行于自动提交状态，状态变化如下：

```
UNLOCKED→PENDING→SHARED→UNLOCKED→PENDING→SHARED→UNLOCKED
```

仅仅是读数据，但在自动提交模式下，却会经历两个加解锁的循环，太麻烦。而且，一个写进程可能插到两个SELECT中间对数据库进行修改，这样，你就不能保证第二次能够读到同样的数据了，而使用BEGIN..COMMIT就可以有此保证。

写事务

下面我们来考虑写数据库，比如UPDATE。和读事务一样，它也会经历UNLOCKED→PENDING→SHARED的变化过程，但接下来就会看到PENDING锁是如何起到关口作用的了。

保留(RESERVED)状态

当一个连接(connection)要向数据库写数据时，从SHARED状态变为RESERVED状态。如果它得到RESERVED锁，也就意味着它已经准备好进行写操作了。即使它没有把修改写入数据库，也可以把修改保存到位于pager的缓冲区中(page cache)了。

当一个连接进入RESERVED状态，pager就开始初始化回卷日志(rollback journal)。回卷日志是一个文件，用于回卷和崩溃恢复，见图5-1。在RESERVED状态下，pager管理着三种页：

(1)已修改的页：包含被B-tree修改的记录，位于page cache中。

(2)未修改的页：包含没有被B-tree修改的记录。

(3)日志页：这是修改页以前的版本，日志页并不存储在page cache中，而是在B-tree修改页之前写入日志。

Page cache非常重要，正是因为它的存在，一个处于RESERVED状态的连接可以真正的开始工作，而不会干扰其它的(读)连接。所以，SQLite可以高效地处理在同一时刻的多个读连接和一个写连接。

未决(PENDING)状态

当一个连接完成修改，需要真正开始提交事务时，执行该过程的pager进入EXCLUSIVE状态。从RESERVED状态开始，pager试着获取PENDING锁，一旦得到，就独占它，不允许任何其它连接获得PENDING锁。既然写操作持有PENDING锁，其它任何连接都不能从UNLOCKED状态进入SHARED状态，即不会再有新的读进程，也不会再有新的写进程。只有那些已经处于SHARED状态的连接可以继续工作。而处于PENDING状态的写进程会一直等到所有这些连接释放它们的锁，然后对数据库加EXCUSIVE锁，进入EXCLUSIVE状态，独占数据库。

排它状态

在EXCLUSIVE状态下，主要的工作是把修改的页从page cache写入数据库文件，这是真正进行写操作的地方。

在pager将修改页写到文件之前，还必须先处理日志。它检查是否所有的日志都写入了磁盘，因为它们可能还位于操作系统的缓冲区中。所以pager得告诉OS把所有的文件写入磁盘，这与synchronous pragma所做的工作相同，如第4章所述。

日志是数据库进行恢复的惟一方法，所以日志对于DBMS非常重要。如果日志页没有完全写入磁盘而发生崩溃，数据库就不能恢复到它原来的状态，此时数据库就处于不一致状态。日志写盘完成后，pager就把所有的修改页写入数据库文件。接下来做什么取决于事务提交的模式，如果是自动提交，那么pager清理日志、page cache，然后由EXCLUSIVE进入UNLOCKED。如果是手动提交，那么pager继续持有EXCLUSIVE锁和回卷日志，直至遇到COMMIT或者ROLLBACK。

总之，出于性能方面的考虑，进程占有排它锁的时间应该尽可能的短，所以DBMS通常都是在真正写文件时才会占有排它锁，这样能大大提高并发性能。

自动提交与效率

调整页缓冲区

回到前面的例子，事务从BEGIN开始，跟着UPDATE。如果在写盘之前，修改操作将缓冲区用完了(也就是说修改操作需要比预设的更多的缓冲区)，这时会发生什么呢？

转换为排它

真正的问题是：到底在哪个(精确的)时刻，到底为什么，pager 从RESERVED转换为EXCLUSIVE。这会发生在两种情况下：当连接到达提交点主动进入排它状态；或页缓冲区已满不得不进入排它状态。

前面我们仅看到了第1种情况，那么，在第2种情况下会发生什么呢？此时pager已不能再存储更多的已修改页，也就不能再做任何修改操作。它必须转换为排它状态，以使工作能够继续进行。实际上也不完全是这样，实际上有软限制和硬限制的区别。

调整页缓冲区的大小

如何决定需要多大的缓冲区尺寸呢？这由你想做什么而定。假设你想修改episodes表的所有记录，那么该表的所有页都会被修改，因此，你就可以计算出episodes表总共需要多少个页并对缓冲区做出调整。可以用sqlite_analyzer到得所有关于episodes表的需要的信息。对每一个表，它都可以做出完备的统计，包括总页数。例如，对于foods数据库，可以得到关于episodes表的如下信息：

```
*** Table EPISODES ****************************************************
Percentage of total database.......... 20.0%
Number of entries..................... 181
Bytes of storage consumed............. 5120
Bytes of payload...................... 3229 63.1%
Average payload per entry............. 17.84
Average unused bytes per entry........ 5.79
Average fanout........................ 4.00
Maximum payload per entry............. 38
Entries that use overflow............. 0 0.0%
Index pages used...................... 1
Primary pages used.................... 4
Overflow pages used................... 0
Total pages used...................... 5
Unused bytes on index pages........... 990 96.7%
Unused bytes on primary pages......... 58 1.4%
Unused bytes on overflow pages........ 0
Unused bytes on all pages............. 1048 20.5%
```

总页数是5，但实际上表只用了4页，还有1页是索引。因为默认的缓冲区大小是2000个页，所以你没有必要担心。在episodes表中有400条记录，也就是说每页可存放约100条记录。所以，在修改所有记录之前你不需要考虑调整页缓冲区，除非episodes中至少有了196000条记录。还要记住，你只需要在有其它连接并发使用数据库的情况下才需要考虑这些，如果只有你自己使用数据库，这些就都不需要考虑了。

等待加锁

我们前面谈到过pager等待从PENDING状态进入EXCLUSIVE状态，那么在这个期间到底发生了什么呢？首先，任何exec()或step()的调用都可能进入等待。当SQLite遇到不能获得锁的情况时，它的默认表现总是向函数返回一个SQLITE_BUSY并使函数继续寻求锁。无论你执行什么命令，都有可能遇到SQLITE_BUSY，包括SELECT命令，都有可能因为有其它的写进程处于未决状态而遇到SQLITE_BUSY。当遇到SQLITE_BUSY时，最简单的选择是重试。但是，下面我们就会看到这并不一定是最好的选择。

使用“忙”句柄

你可以使用一个忙句柄，而不是一遍遍地重试。忙句柄是一个函数，你创建它用来消磨时间或做其它任何事情——如给岳母发一封邮件(？)。它仅在SQLite不能获得锁时被调用。忙句柄必须做的唯一的事是返回一个值，告诉SQLite下一步该做什么。如果它返回TRUE，SQLite将会继续尝试获得锁；如果它返回FALSE，SQLite将向申请锁的函数返回SQLITE_BUSY。看下面的例子：

```
counter = 1
def busy()
    counter = counter + 1
    if counter == 2
        return 0
    end
    spam_mother_in_law(100)
    return 1
end
db.busy_handler(busy)
stmt = db.prepare('SELECT * FROM episodes;')
stmt.step()
stmt.finalize()
```

spam_mother_in_law()完成一个发邮件功能。

step()函数必须获得一个SHARED锁以完成SELECT操作。如果此时有一个写进程活动，正常情况下step()会返回SQLITE_BUSY。但是，在上面程序中却不是这样，而是由pager调用busy()函数，因为它已经被注册队忙句柄。busy()函数增加计数，给你岳母发一封邮件，并且返回1，在pager中会被翻译成true——继续申请锁。Pager再次申请获得SHARED锁，但数据库仍然被锁着，于是pager再次调用busy()函数。只有此时，busy()函数返回0，在pager中会被翻译成false——返回SQLITE_BUSY。

使用正确的事务

编码

现在，你对API、事务和锁已经有了很好的了解了。最后，我们把这3个内容在代码中结合到一起。

使用多个连接

如果你曾经为其它的关系型数据库编写过程序，你就会发现有些适用于那些数据库的方法不一定适用于SQLite。使用其它数据库时，经常会在同一个代码块中打开多个连接，典型的例子就是在一个连接中返复遍历一个表而在另一个连接中修改它的记录。

在SQLite中，在同一个代码块中使用多个连接会引起问题，必须小心地对待这种情况。请看下面代码：

```
c1 = open('foods.db')
c2 = open('foods.db')
stmt = c1.prepare('SELECT * FROM episodes')
while stmt.step()
    print stmt.column('name')
    c2.exec('UPDATE episodes SET …)
end
stmt.finalize()
c1.close()
c2.close()
```

问题很明显，当c2试图执行UPDATE时，c1拥有一个SHARED锁，这个锁只有等stmt.finalize()之后才会释放。所以，是不可能成功写数据库的。最好的办法是在一个连接中完成工作，并且在同一个BEGIN IMMEDIATE事务中完成。新程序如下：

```
c1 = open('foods.db')
# Keep trying until we get it
while c1.exec('BEGIN IMMEDIATE') != SQLITE_OK
end
stmt = c1.prepare('SELECT * FROM episodes')
while stmt.step()
    print stmt.column('name')
    c1.exec('UPDATE episodes SET …)
end
stmt.finalize()
c1.exec('COMMIT')
c1.close()
```

在这种情况下，你应该在单独的连接中使用语句(statement)来完成读和写，这样，你就不必担心数据库锁会引发问题了。但是，这个特别的示例仍然不能工作。如果你在一个语句(statement)中返复遍历一个表而在另一个语句中修改它的记录，还有一个附加的锁问题你需要了解，我们将在下面介绍。

表锁

即使只使用一个连接，在有些边界情况下也会出现问题。不要认为一个连接中的两个语句(statements)就能协调工作，至少有一个重要的例外。

当在一个表上执行了SELECT命令，语句对象会在表上创建一个B-tree游标。如果表上有一个活动的B-tree游标，即使是本连接中的其它语句也不能够再修改这个表。如果做这种尝试，将会得到SQLITE_BUSY。看下面的例子：

```
c = sqlite.open("foods.db")
stmt1 = c.compile('SELECT * FROM episodes LIMIT 10')
while stmt1.step() do
    # Try to update the row
    row = stm1.row()
    stmt2 = c.compile('UPDATE episodes SET …')
    # Uh oh: ain't gonna happen
    stmt2.step()
end
stmt1.finalize()
stmt2.finalize ()
c.close()
```

这里我们只使用了一个连接。但当调用stmt2.step()则不会工作，因为stmt1拥有episodes表的一个游标。在这种情况下，stmt2.step()有可能成功地将锁升级到EXCLUSIVE，但仍会返回SQLITE_BUSY，因为episodes的游标会阻止它修改表。完成这种操作有两种方法：

- 遍历一个语句的结果集，在内存中保存需要的信息。定案这个读语句，然后执行修改操作。
- 将SELECT的结果存到一个临时表中并用读游标打开它。这时同时有一个读语句和一个写语句，但它们在不同的表上，所以不会影响主表上的写操作。写完成后，删掉临时表就是了。

当表上打开了一个语句，它的B-tree游标在两种情况下会被移除：

- 到达了语句结果集的尾部。这时step()会自动地关闭语句的游标。从VDBE的角度，当到达结果集的尾部时，CDBE遇到Close命令，这将导致所有相关游标的关闭。
- 程序显式地调用了finalize()，所有相关游标将关闭。

在很多编程语言扩展中，statement对象的close()函数会自动调用sqlite3_finalize()。

有趣的临时表

临时表使你可以做到不违反规则。如果你确实需要在一个代码块中使用两个连接，或者使用两个语句(statement)操作同一个表，你可以安全地在临时表上如此做。当一个连接创建了一个临时表，不需要得到RESERVED锁，因为临时表存在于数据库文件之外。有两种方法可以做到这一点，看你想如何管理并发。请看如下代码：

```
c1 = open('foods.db')
c2 = open('foods.db')
c2.exec('CREATE TEMPORARY TABLE temp_epsidodes as SELECT * from episodes')
stmt = c1.prepare('SELECT * FROM episodes')
while stmt.step()
    print stmt.column('name')
    c2.exec('UPDATE temp_episodes SET …')
end
stmt.finalize()
c2.exec('BEGIN IMMEDIATE')
c2.exec('DELETE FROM episodes')
c2.exec('INSERT INTO episodes SELECT * FROM temp_episodes')
c2.exec('COMMIT')
c1.close()
c2.close()
```

上面的例子可以完成功能，但不好。episodes表中的数据要全部删除并重建，这将丢失episodes表中的所有完整性约束和索引。下面的方法比较好：

```
c1 = open('foods.db')
c2 = open('foods.db')
c1.exec('CREATE TEMPORARY TABLE temp_episodes as SELECT * from episodes')
stmt = c1.prepare('SELECT * FROM temp_episodes')
while stmt.step()
    print stmt.column('name')
    c2.exec('UPDATE episodes SET …') # What about SQLITE_BUSY?
end
stmt.finalize()
c1.exec('DROP TABLE temp_episodes')
c1.close()
c2.close()
```

定案的重要性

使用SELECT语句必须要意识到，其SHARED锁(大多数时候)直到finalize()被调用后才会释放，请看下面代码：

```
stmt = c1.prepare('SELECT * FROM episodes')
while stmt.step()
    print stmt.column('name')
end
c2.exec('BEGIN IMMEDIATE; UPDATE episodes SET …; COMMIT;')
stmt.finalize()
```

如果你用C API写了与上例等价的程序，它实际上是能够工作的。尽管没有调用finalize()，但第二个连接仍然能够修改数据库。在告诉你为什么之前，先来看第二个例子：

```
c1 = open('foods.db')
c2 = open('foods.db')
stmt = c1.prepare('SELECT * FROM episodes')
stmt.step()
stmt.step()
stmt.step()
c2.exec('BEGIN IMMEDIATE; UPDATE episodes SET …; COMMIT;')
stmt.finalize()
```

假设episodes中有100条记录，程序仅仅访问了其中的3条，这时会发生什么情况呢？第2个连接会得到SQLITE_BUSY。

在第1个例子中，当到达语句结果集尾部时，会释放SHARED锁，尽管还没有调用finalize()。在第2个例子中，没有到达语句结果集尾部，SHARED锁没有释放。所以，c2不能执行UPDATE操作。

这个故事的中心思想是：不要这么做，尽管有时这么做是可以的。在用另一个连接进行写操作之前，永远要先调用finalize()。

共享缓冲区模式

现在你对并发规则已经很清楚了，但我还要找些事来扰乱你。SQLite提供一种可选的并发模式，称为共享缓冲区模式，它允许在单一的线程中操作多个连接。

在共享缓冲区模式中，一个线程可以创建多个连接来共享相同的页缓冲区。进而，这组连接可以有多个"读"和一个"写"同时工作于相同的数据库。缓冲区不能在线程间共享，它被严格地限制在创建它的线程中。因此，"读"和"写"就需要准备处理与表锁有关的一些特殊情况。

当 readers读表时， SQLite自动在这些表上加锁，writer就不能再改这些表了。如果writer试图修改一个有读锁的表，会得到SQLITE_LOCKED。如果readers运行在read-uncommitted模式(通过read_uncommitted pragma来设置)，则当readers读表时，writer也可以写表。在这种情况下，SQLite不为readers所读的表加读锁，结果就是readers和writer互不干扰。也因此，当一个writer修改表时，这些readers可能得到不一致的结果。

第6章 核心C API

本章介绍用于数据库操作的SQLite API。第5章已经介绍了API如何工作，本章关注细节。

本章从几个例子开始，深入介绍C API。学完本章之后，你会看到每个C API函数都与常用的数据库操作有关，包括执行命令、管理事务、取记录、处理错误等等。

SQLite的版本3的API包括大约80个函数。只有8个函数在连接、查询和断开连接时是必须的，其它的函数用来完成特定的任务。

如第5章所述，版本3与2的API相比有较大改变。最值得关注的一个改变是增加了对UTF的支持。所有接受字符串做为参数或返回字符串的函数都同时具有UTF-8和UTF-16的相似体。例如，sqlite3_open()，接受一个UTF-8的数据库名做参数；而sqlite3_open16()具有同样的功能与格式，但参数使用UTF-16编码。本章一般只介绍UTF-8的函数，UTF-16版本仅仅是在名字上有微小差别。

本章最好顺序地读，如果在细节上有问题，可以参考附录B。

空注：第6、7两章应该也是本书的精华了，主要介绍对SQLite进行编程的方法。大多数SQLite的使用者可能更关心这两章，但我又不开发基于SQLite的应用程序，研究SQLite纯粹出于兴趣，个人更关心SQLite本身的实现方法，所以对这部分内容只是略做浏览。关心这部分内容的兄弟还是得自己看原文。

封装的查询

你已经熟悉了SQLite执行查询的方法，包括在一个单独的函数中执行封装的SQL。我们从封装的SQL开始介绍，因为这些函数简单、独立且易用。它们是好的起点，使你能得到乐趣，又不会被过多的细节所困扰。

连接和断开连接

在执行SQL命令之前，首先要连接数据库。因为SQLite数据库存储在一个单独的操作系统文件当中，所以连接数据库可以理解为“打开”数据库。同样，断开连接也就是关闭数据库。

打开数据库用sqlite3_open()或sqlite3_open16()函数，它们的声明如下：

```
int sqlite3_open(
    const char *filename,           /* Database filename (UTF-8) */
    sqlite3 **ppDb                    /* OUT: SQLite db handle */
);
int sqlite3_open16(
    const void *filename,            /* Database filename (UTF-16) */
    sqlite3 **ppDb                    /* OUT: SQLite db handle */
);
```

其中，filename参数可以是一个操作系统文件名，或字符串':memory:'，或一个空指针(NULL)。用后两者将创建内存数据库。如果filename不为空，先尝试打开，如果文件不存在，则用这个名字创建一个新的数据库。

关闭连接使用sqlite3_close()函数，它的声明如下：

```
int sqlite3_close(sqlite3*);
```

为了sqlite3_close()能够成功执行，所有与连接所关联的已编译的查询必须被定案。如果仍然有查询没有定案，sqlite3_close()将返回SQLITE_BUSY和错误信息：Unable to close due to unfinalized statements。

执行Query

函数sqlite3_exec()提供了一种执行SQL命令的快速、简单的方法，它特别适合处理对数据库的修改操作(不需要返回数据)。sqlite3_exec()的声明如下：

```
int sqlite3_exec(

    sqlite3*, /* An open database */
    const char *sql, /* SQL to be executed */
    sqlite_callback, /* Callback function */
    void *data /* 1st argument to callback function */
    char **errmsg /* Error msg written here */
);
```

SQL命令由sql参数提供，它可以由多个SQL命令构成，sqlite3_exec()会对其中每个命令进行分析并执行，直到命令串结束或遇到一个错误。列表6-1(来自create.c)说明了sqlite3_exec()的用法：

列表6-1 对简单的命令使用sqlite3_exec()

```
#include <stdio.h>
#include <stdlib.h>
#include "util.h"
#pragma comment(lib, "sqlite3.lib")
int main(int argc, char **argv)
{
    sqlite3 *db;
    char *zErr;
    int rc;
    char *sql;
    rc = sqlite3_open("test.db", &db);
    if (rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
    sql = "create table episodes( id integer primary key,"
        "                   name text, cid int)";
    rc = sqlite3_exec(db, sql, NULL, NULL, &zErr);
    if (rc != SQLITE_OK) {
        if (zErr != NULL) {
            fprintf(stderr, "SQL error: %s\n", zErr);
            sqlite3_free(zErr);
        }
    }
    sql = "insert into episodes (name,id) values ('Cinnamon Babka2',1)";
    rc = sqlite3_exec(db, sql, NULL, NULL, &zErr);
    if (rc != SQLITE_OK) {
        if (zErr != NULL) {
            fprintf(stderr, "SQL error: %s\n", zErr);
            sqlite3_free(zErr);
        }
    }
    sqlite3_close(db);
    return 0;
}
```

处理记录

如第5章所述，还是有可能从sqlite3_exec()取得记录的。sqlite3_exec()包含一个回叫(callback)机制，提供了一种从SELECT语句得到结果的方法。这个机制由sqlite3_exec()函数的第3和第4个参数实现。第3个参数是一个指向回叫函数的指针，如果提供了回叫函数，SQLite则会在执行SELECT语句期间在遇到每一条记录时调用回叫函数。回叫函数的声明如下：

```
typedef int (*sqlite3_callback)(
    void*, /* Data provided in the 4th argument of sqlite3_exec() */
    int, /* The number of columns in row */
    char**, /* An array of strings representing fields in the row */
    char** /* An array of strings representing column names */
);
```

函数sqlite3_exec()的第4个参数是一个指向任何应用程序指定的数据的指针，这个数据是你准备提供给回叫函数使用的。SQLite将把这个数据作为回叫函数的第1个参数传递。

总之，sqlite3_exec()允许你处理一批命令，并且你可以使用回叫函数来收集所有返回的数据。例如，先向episodes表插入一条记录，再从中查询所有记录，所有这些都在一个sqlite3_exec()调用中完成。完整的程序代码见列表6-2，它来自exec.c。

列表6-2 将sqlite3_exec()用于记录处理

```
#include <stdio.h>
#include <stdlib.h>
#include "util.h"
#pragma comment(lib, "sqlite3.lib")
int callback(void* data, int ncols, char** values, char** headers);
int main(int argc, char **argv)
{
    sqlite3 *db;
    int rc;
    char *sql;
    char *zErr;
       char* data;
    rc = sqlite3_open("test.db", &db);
    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
    data = "Callback function called";
    sql = "insert into episodes (name, cid) values ('Mackinaw Peaches', 1);"
          "select * from episodes;";
    rc = sqlite3_exec(db, sql, callback, data, &zErr);
    if(rc != SQLITE_OK) {
        if (zErr != NULL) {
            fprintf(stderr, "SQL error: %s\n", zErr);
            sqlite3_free(zErr);
        }
    }
    sqlite3_close(db);
    return 0;
}
int callback(void* data, int ncols, char** values, char** headers)
{
    int i;
    fprintf(stderr, "%s: ", (const char*)data);
    for(i=0; i < ncols; i++) {
        fprintf(stderr, "%s=%s ", headers[i], values[i]);
    }
    fprintf(stderr, "\n");
    return 0;
}
```

字符串处理

```
int main(int argc, char **argv)
{
    char *sql;
    char *trouble = "'Here's trouble'";
    sql = sqlite3_mprintf("insert into x values('%q')", trouble);
    fprintf(stdout, "%s\n", sql);
    sqlite3_free(sql);
    return 0;
}
```

The result sql will contain

insert into x values('''Here''s trouble''')

Listing 6-3. Using sqlite3_vmprintf()

```
int execute(sqlite3 *db, const char* sql, ...)
{
    char *err, *tmp;
    va_list ap;
    va_start(ap, sql);
    tmp = sqlite3_vmprintf(sql, ap);
    va_end(ap);
    int rc = sqlite3_exec(db, tmp, NULL, NULL, &err);
    if(rc != SQLITE_OK) {
        if (err != NULL) {
            fprintf(stdout, "execute() : Error %i : %s\n", rc, err);
            sqlite3_free(err);
        }
    }
    sqlite3_free(tmp);
    return rc;
}
```

Get Table查询

```
int sqlite3_get_table(
    sqlite3*, /* An open database */
    const char *sql, /* SQL to be executed */
    char ***resultp, /* Result written to a char *[] that this points to */
    int *nrow, /* Number of result rows written here */
    int *ncolumn, /* Number of result columns written here */
    char **errmsg /* Error msg written here */
);
```

Listing 6-4. Using sqlite3_get_table

```
void main(int argc, char **argv)
{
    sqlite3 *db;
    char *zErr;
    int rc,i;
    char *sql;
    char **result;
       int nrows, ncols;
    /* Connect to database, etc. */
    rc = sqlite3_open("test.db", &db);
    sql = "select * from episodes;";
    rc = sqlite3_get_table(db, sql, &result, &nrows, &ncols, &zErr);
    /* Do something with data */
       printf("rows=%d,cols=%d\n",nrows,ncols);
       for (i=0;i<=nrows;i++)
              printf("%-5s%-20s%-5s\n",result[3*i],result[3*i+1],result[3*i+2]);
    /* Free memory */
    sqlite3_free_table(result);
}
```

If, for example, the result set returned is of the form

rows=2,cols=3

```
id   name                cid
1    Cinnamon Babka2     (null)
2    Mackinaw Peaches    1
```

预处理的查询

As you’ll recall from Chapter 5, prepared queries are performed in three basic steps: compilation, execution, and finalization. This process is illustrated in Figure 6-1.

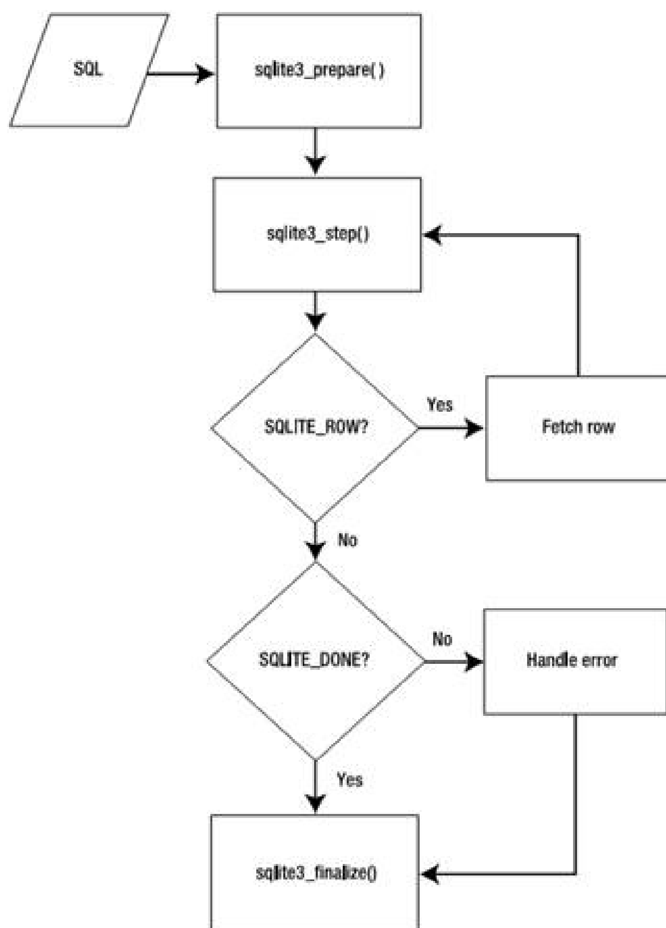


Figure 6-1. Prepared query processing

Now that you’ve seen the whole process, let’s go through an example. A simple, complete program using a prepared query is listed in Listing 6-6. It is taken from select.c in the examples.

Listing 6-6. Using Prepared Queries

```
#include <string.h>
int main(int argc, char **argv)
{
    int rc, i, ncols;
    sqlite3 *db;
    sqlite3_stmt *stmt;
    char *sql;
    const char *tail;
    rc = sqlite3_open("test.db", &db);
    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
    sql = "select * from episodes;";
    rc = sqlite3_prepare(db, sql, (int)strlen(sql), &stmt, &tail);
    if(rc != SQLITE_OK) {
        fprintf(stderr, "SQL error: %s\n", sqlite3_errmsg(db));
    }
    rc = sqlite3_step(stmt);
    ncols = sqlite3_column_count(stmt);
    while(rc == SQLITE_ROW) {
        for(i=0; i < ncols; i++) {
            fprintf(stderr, "'%s' ", sqlite3_column_text(stmt, i));
        }
        fprintf(stderr, "\n");
        rc = sqlite3_step(stmt);
    }
    sqlite3_finalize(stmt);
    sqlite3_close(db);
    return 0;
}
```

跟sqlite3_exec()相似，sqlite3_prepare()也可以接受一个包括多个SQL语句的字符串。不同的是sqlite3_prepare()只处理字符串中的第1个语句。But it does make it easy for you to process subsequent SQL statements in the string by providing the pzTailout parameter. After you call sqlite3_prepare(), it will point this parameter (if provided) to the starting position of the next statement in the zSQL string. Using pzTail, processing a batch of SQL commands in a given string can be executed in a loop as follows:

```
while(sqlite3_complete(sql) {
    rc = sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);
    /* Process query results */
    /* Skip to next command in string. */
    sql = tail;
}
```

取记录

取字段信息

你可以使用sqlite3_column_name()来取得各字段的名称：

```
    const char *sqlite3_column_name( sqlite3_stmt*, /* statement handle */
        int iCol /* column ordinal */);
```

类似地，你可以使用sqlite3_column_type()取得各字段的存储类：

```
    int sqlite3_column_type( sqlite3_stmt*, /* statement handle */
        int iCol /* column ordinal */);
```

这个函数返回一个整数值，代表5个存储类的代码，定义如下：

```
#define SQLITE_INTEGER 1
#define SQLITE_FLOAT 2
#define SQLITE_TEXT 3
#define SQLITE_BLOB 4
#define SQLITE_NULL 5
```

这些是SQLite本身的类型，或称存储类，在第4章有详细介绍。All data stored within a SQLite database is stored in one of these five forms, depending on its initial representation and the affinity of the column. For our purposes, the terms storage class and data type are synonymous. For more information on storage classes, see the sections “Storage Classes” and “Type Affinity” in Chapter 4.

你可以使用sqlite3_column_decltype()函数获得字段声明的数据类型：

```
    const char *sqlite3_column_decltype( sqlite3_stmt*, /* statement handle */
        int /* column ordinal */);
```

如果结果集中的一列不是来自一个实际的字段(如来自于表达式、函数或聚合的结果)，这个函数将返回NULL。For example, suppose you have a table in your database defined as

```
CREATE TABLE t1(c1 INTEGER);
```

Then you execute the following query:

```
SELECT c1 + 1, 0 FROM t1;
```

In this case, sqlite3_column_decltype() will return INTEGER for the first column and NULL for

the second.

还可以用下列函数获得字段的其它信息：

```
const char *sqlite3_column_database_name(sqlite3_stmt *pStmt, int iCol);

const char *sqlite3_column_table_name(sqlite3_stmt *pStmt, int iCol);
const char *sqlite3_column_origin_name(sqlite3_stmt *pStmt, int iCol);
```

The first function will return the database associated with a column, the second its table, and

the last function returns the column's actual name as defined in the schema. That is, if you

assigned the column an alias in the SQL statement, sqlite3_column_origin_name() will return

its actual name as defined in the schema. Note that these functions are only available if you

compile SQLite with the SQLITE_ENABLE_COLUMN_METADATA preprocessor directive.

列元数据：

字段的详细信息可以从一个独立的query获得，使用sqlite3_table_column_metadata()函数，声明如下：

```
SQLITE_API int sqlite3_table_column_metadata(
  sqlite3 *db,                /* Connection handle */
  const char *zDbName,        /* Database name or NULL */
  const char *zTableName,     /* Table name */
  const char *zColumnName,    /* Column name */
  char const **pzDataType,    /* OUTPUT: Declared data type */
  char const **pzCollSeq,     /* OUTPUT: Collation sequence name */
  int *pNotNull,              /* OUTPUT: True if NOT NULL constraint exists */
  int *pPrimaryKey,           /* OUTPUT: True if column part of PK */
  int *pAutoinc               /* OUTPUT: True if column is auto-increment */
);
```

这个函数包含输入和输出参数。它不在statement句柄下工作，但需要提供连接句柄、数据库名、表名和列名。可选的数据库名指明附加的逻辑数据库名(一个连接上可能附加多个数据库)。表名和字段名是必须的。

取字段值

可以使用sqlite3_column_xxx()函数取当前记录中每个字段的值，其一般形式为：

```
xxx sqlite3_column_xxx( sqlite3_stmt*, /* statement handle */
    int iCol /* column ordinal */);
```

xxx表示你希望得到的数据类型。sqlite3_column_xxx()包括以下函数：

```
int sqlite3_column_int(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
long long int sqlite3_column_int64(sqlite3_stmt*, int iCol);
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
```

对每个函数，SQLite都会将字段值从存储类转化为函数指定的结果类型。Table 6-1中是转换规则。

Table 6-1. Column Type Conversion Rules

```
Internal Type          Requested Type            Conversion
NULL                    INTEGER                   Result is 0\.
NULL                    FLOAT                        Result is 0.0\.
NULL                    TEXT                         Result is a NULL pointer.
```

一个实际的例子

To help solidify all of these column functions, Listing 6-7 (taken from columns.c) illustrates using the functions we've described to retrieve column information and values for a simple SELECT statement.

Listing 6-7. Obtaining Column Information

```
#include <string.h>
int main(int argc, char **argv)
{
    int rc, i, ncols, id, cid;
       char *name, *sql;
    sqlite3 *db; sqlite3_stmt *stmt;
       sql = "select id,cid,name from episodes";
    sqlite3_open("test.db", &db);
    sqlite3_prepare(db, sql, strlen(sql), &stmt, NULL);
    ncols = sqlite3_column_count(stmt);
    rc = sqlite3_step(stmt);
    /* Print column information */
    for(i=0; i < ncols; i++) {
        fprintf(stdout, "Column: name=%s, storage class=%i, declared=%s\n",
                         sqlite3_column_name(stmt, i),
                         sqlite3_column_type(stmt, i),
                         sqlite3_column_decltype(stmt, i));
    }
    fprintf(stdout, "\n");
    while(rc == SQLITE_ROW) {
        id = sqlite3_column_int(stmt, 0);
        cid = sqlite3_column_int(stmt, 1);
        name = (char *)sqlite3_column_text(stmt, 2);
        if(name != NULL){
            fprintf(stderr, "Row:  id=%i, cid=%i, name='%s'\n", id,cid,name);
        } else {
            /* Field is NULL */
            fprintf(stderr, "Row:  id=%i, cid=%i, name=NULL\n", id,cid);
        }
        rc = sqlite3_step(stmt);
    }
    sqlite3_finalize(stmt);
    sqlite3_close(db);
    return 0;
}
```

参数化的查询

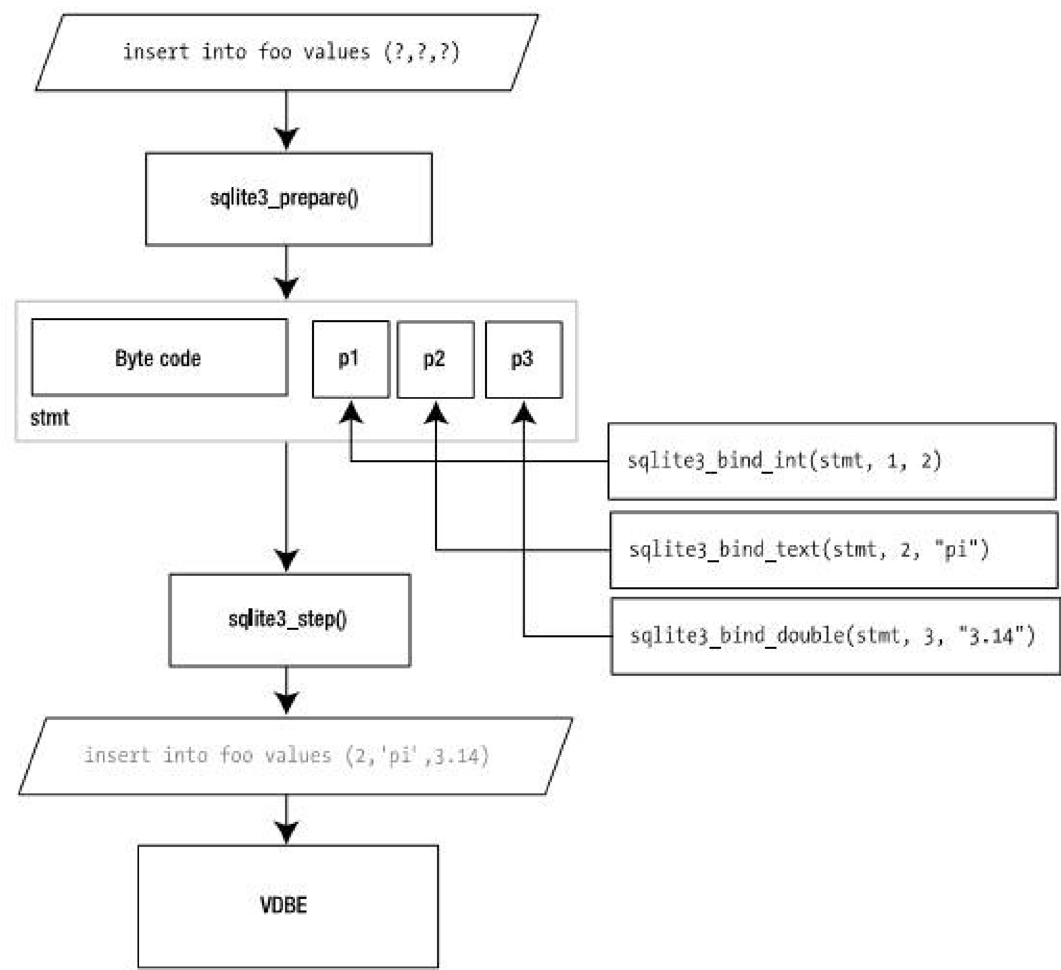


Figure 6-2. SQL parameter binding

错误和意外

有些API是很可能出错的，在编码时总要记得3件事：错误、忙状态和schema改变。

处理错误

很多API函数返回整数结果码，这表示它们可以返回错误码。在使用一个函数之前，应该仔细阅读关于该函数的说明(见附录B)，看它可能引发什么错误。API中定义了大约23种错误。所有的SQLite返回码见表6-2。所有能够返回这些码的函数包括：

```
sqlite3_bind_xxx()
sqlite3_close()
sqlite3_create_collation()
sqlite3_collation_needed()
sqlite3_create_function()
sqlite3_prepare()
sqlite3_exec()
sqlite3_finalize()
sqlite3_get_table()
sqlite3_open()
sqlite3_reset()
sqlite3_step()
sqlite3_transfer_bindings()
```

可以使用函数sqlite3_errmsg()获得附加的错误信息，其声明如下：

```
const char *sqlite3_errmsg(sqlite3 *);
```

它以一个连接句柄作参数，返回该连接最近的一条错误信息。如果还没有发生错误，它返回“not an error”。

表6-2 SQLit的返回码

返回码	说明
SQLITE_OK	The operation was successful.
SQLITE_ERROR	General SQL error or missing database. It may be possible to obtain more error information depending on the error condition (SQLITE_SCHEMA, for example).
SQLITE_PERM	Access permission denied. Cannot read or write to the database file.
SQLITE_ABORT	A callback routine requested an abort.
SQLITE_BUSY	The database file is locked.
SQLITE_LOCKED	A table in the database is locked.
SQLITE_NOMEM	A call to malloc() has failed within a database operation.
SQLITE_READONLY	An attempt was made to write to a read-only database.
SQLITE_INTERRUPT	Operation was terminated by sqlite3_interrupt().
SQLITE_IOERR	Some kind of disk I/O error occurred.
SQLITE_CORRUPT	The database disk image is malformed. This will also occur if an attempt is made to open a non-SQLite database file as a SQLite database. SQLITE_FULL Insertion failed because the database is full. There is no more space on the file system or the database file cannot be expanded.
SQLITE_CANTOPEN	SQLite was unable to open the database file.
SQLITE_PROTOCOL	The database is locked or there has been a protocol error.

SQLITE_EMPTY	(Internal only) The database table is empty.
SQLITE_SCHEMA	The database schema has changed.
SQLITE_CONSTRAINT	Abort due to constraint violation. This constant is returned if the SQL statement would have violated a database constraint (such as attempting to insert a value into a unique index that already exists in the index).
SQLITE_MISMATCH	Data type mismatch. An example of this is an attempt to insert non-integer data into a column labeled INTEGER PRIMARY KEY. For mostcolumns, SQLite ignores the data type and allows any kind of data to be stored. But an INTEGER PRIMARY KEY column is only allowed to store integer data.
SQLITE_MISUSE	Library was used incorrectly. This error might occur if one or more of the SQLite API routines is used incorrectly. Examples of incorrect usage include calling sqlite3_exec() after the database has been closed using sqlite3_close() or calling sqlite3_exec() with the same database pointer simultaneously from two separate threads.
SQLITE_NOLFS	Uses OS features not supported on host. This value is returned if the SQLite library was compiled with large file support (LFS) enabled butLFS isn't supported on the host operating system.
SQLITE_AUTH	Authorization denied. This occurs when a callback function installed using sqlite3_set_authorizer() returns SQLITE_DENY.
SQLITE_ROW	sqlite3_step() has another row ready.
SQLITE_DONE	sqlite3_step() has finished executing.

处理忙状态

操作控制

API提供了几个函数，可以用来监视或管理编译期间和运行时的SQL命令。这些函数允许你建立回叫函数，并以此对不同的数据库事件进行监视和控制(当事件发生时)。

提交Hook函数

使用sqlite3_commit_hook()函数可以在特定连接提交事务时对其进行监视。其声明如下：

```
void *sqlite3_commit_hook( sqlite3 *cnx, /* database handle */
    int(*xCallback)(void *data), /* callback function */
    void *data); /* application data */
```

回卷Hook函数

回卷Hook函数与提交Hook函数相类似，但它在特定连接回卷事务时对其进行监视。

```
void *sqlite3_rollback_hook(sqlite3 *cnx, void(*xCallback)(void *data), void *data);
```

修改Hook函数

函数sqlite3_update_hook()用来监视特定数据库连接所有的UPDATE、INSERT和DELETE操作，对这些操作中所涉及的每一行都进行监视，其声明如下：

```
void *sqlite3_update_hook(
    sqlite3 *cnx,
    void(*)(void *, int, char const*, char const*, sqlite_int64),
    void *data);
```

The first argument of the callback function is a pointer to application-specific data, which you provide in the third argument. The callback function has the following form:

```
void callback ( void * data,
    int operation_code,
    char const *db_name,
    char const *table_name,
    sqlite_int64 rowid),
```

授权函数

sqlite3_set_authorizer()是最强有力的事件过滤函数。用它可以在查询编译的时候对其进行监视和控制。其声明如下：

```
int sqlite3_set_authorizer(
    sqlite3*,
    int (*xAuth)( void*,int,
        const char*, const char*,
        const char*,const char*),
    void *pUserData
    );
```

其中注册了一个callback函数，作为授权函数。SQLite在一些数据库事件的命令编译阶段将会调用它(不是在执行阶段)。这个函数的用意是使用SQLite能够安全地执行用户提供的SQL(user-supplied SQL)。它提供了一种途径将这类SQL限制在特定的操作上或拒绝对某些

表或字段的存取。

Callback的声明形式如下：

```
int auth( void*, /* user data */
    int, /* event code */
    const char*, /* event specific */
    const char*, /* event specific */
    const char*, /* database name */
    const char* /* trigger or view name */ );
```

第1个参数是一个数据指针，它会传递给sqlite3_set_authorizer()函数的第4个参数。第2个参数是一个常量，可选值在表6-3中列出。这些常量值表示需要授权的是什么操作。第3、4个函数的含义决定于事件代码(第2个参数，参表6-3)。

第5个参数是数据库名。第6个参数是最内层触发器或视图的名称，就是这个触发器或视图企图存取数据库。如果这个参数为NULL，则说明这种存取的企图是直接由顶层的SQL引发的。

授权函数的返回值应该是SQLITE_OK、SQLITE_DENY或SQLITE_IGNORE之一。前两个值的含义对所有事件都是确定的——接受或拒绝SQL。SQLITE_DENY将会取消整个SQL语句的执行并生成一个错误。

SQLITE_IGNORE的含义与事件有关。如果SQL语句是读或改记录，会在语句试图操作的每个字段上产生SQLITE_READ或SQLITE_UPDATE事件。在这种情况下，如果回叫函数返回SQLITE_IGNORE，这些字段将从操作中被排除(高：别的字段继续操作，这些字段就不操作了)。具体说，试图读的返回NULL，试图写的则什么也不做(silently fail)。

表6-3 SQLite的授权事件

事件代码	参数3	参数4
SQLITE_CREATE_INDEX	Index name	Table name
SQLITE_CREATE_TABLE	Table name	NULL
SQLITE_CREATE_TEMP_INDEX	Index name	Table name
SQLITE_CREATE_TEMP_TABLE	Table name	NULL
SQLITE_CREATE_TEMP_TRIGGER	Trigger name	Table name
SQLITE_CREATE_TEMP_VIEW	View name	NULL
SQLITE_CREATE_TRIGGER	Trigger name	Table name
SQLITE_CREATE_VIEW	View name	NULL
SQLITE_DELETE	Table name	NULL
SQLITE_DROP_INDEX	Index name	Table name
SQLITE_DROP_TABLE	Table name	NULL
SQLITE_DROP_TEMP_INDEX	Index name	Table name
SQLITE_DROP_TEMP_TABLE	Table name	NULL
SQLITE_DROP_TEMP_TRIGGER	Trigger name	Table name
SQLITE_DROP_TEMP_VIEW	View name	NULL
SQLITE_DROP_TRIGGER	Trigger name	Table name
SQLITE_DROP_VIEW	View name	NULL
SQLITE_INSERT	Table name	NULL
SQLITE_PRAGMA	Pragma name	First argument or NULL
SQLITE_READ	Table name	Column name
SQLITE_SELECT	NULL	NULL
SQLITE_TRANSACTION	NULL	NULL
SQLITE_UPDATE	Table name	Column name
SQLITE_ATTACH	Filename	NULL
SQLITE_DETACH	Database name	NULL

下面例子说明授权函数的使用(完整的程序在authorizer.c中)。

这是一个很长的例子，会用授权函数对很多不同的数据库事件进行过滤，所以我们通过程序片段来进行说明。见列表6-10。

列表6-10 授权函数示例

授权函数的一般形式为：

```
int auth( void* x, int type,
          const char* a, const char* b,
          const char* c, const char* d )
{
    const char* operation = a;
    //printf( "    %s ", event_description(type));
    /* Filter for different database events
    ** from SQLITE_TRANSACTION to SQLITE_INSERT,
    ** UPDATE, DELETE, ATTACH, etc. and either allow or deny
    ** them.
    */
    return SQLITE_OK;
}
```

授权函数做的第1件事是：看看事务状态是否改变；如果改变，则输出一个信息：

```
    if((a != NULL) && (type == SQLITE_TRANSACTION)) {
        printf(": %s êÂÎñ", operation);
    }
```

下一步是对引起schema改变的事件进行过滤：

```
    switch(type) {
        case SQLITE_CREATE_INDEX:
        case SQLITE_CREATE_TABLE:
        case SQLITE_CREATE_TRIGGER:
        case SQLITE_CREATE_VIEW:
        case SQLITE_DROP_INDEX:
        case SQLITE_DROP_TABLE:
        case SQLITE_DROP_TRIGGER:
        case SQLITE_DROP_VIEW:
        {
            printf(": Schema¸Ä±äáË¡£");
        }
    }
```

下一步是对读的企图进行检查，这种企图是基于字段的。这里，所有的读都被允许，除了z字段。当要读z字段时，函数返回SQLITE_IGNORE，这将导致SQLite在读这个字段时返回NULL，从而有效地保护其数据。

```
    if(type == SQLITE_READ) {
        printf(": Read of %s.%s ", a, b);
        /* Block attempts to read column z */
        if(strcmp(b,"z")==0) {
            printf("-> DENIED\n");
            return SQLITE_IGNORE;
        }
    }
```

下面是INSERT和UPDATE的过滤。所有的插入被允许。对x字段的修改被拒绝。这样不会锁住UPDATE的执行，而是简单地过滤掉对x字段的修改企图。

```
    if(type == SQLITE_INSERT) {
        printf(": 2åèë¼ÇÂ¼ into %s ", a);
    }
    if(type == SQLITE_UPDATE) {
        printf(": Update of %s.%s ", a, b);
        /* Block updates of column x */
        if(strcmp(b,"x")==0) {
            printf("-> DENIED\n");
            return SQLITE_IGNORE;
        }
    }
```

最后，对DELETE、ATTACH和DETACH进行过滤，在遇到这些事件时只是简单地给出通知。

```
    if(type == SQLITE_DELETE) {
        printf(": Delete from %s ", a);
    }
    if(type == SQLITE_ATTACH) {
        printf(": %s", a);
    }
    if(type == SQLITE_DETACH) {
        printf("-> %s", a);
    }
```

下面是主程序，为了介绍的方便，也会分成多个片段。

```
int main(int argc, char **argv)
{
    sqlite3 *db, *db2;
    char *zErr;
    int rc;
    /* -------------------------------------------------------------------------
    **  Setup
    ** -------------------------------------------------------------------------
    */
    /* Connect to test.db */
    rc = sqlite3_open("test.db", &db);
    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
    /* -------------------------------------------------------------------------
    **  Authorize and test
    ** -------------------------------------------------------------------------
    */
    /* Register the authorizer function */
    sqlite3_set_authorizer(db, auth, NULL);
    /* Test transactions events */
    printf("program : Starting transaction\n");
    sqlite3_exec(db, "BEGIN", NULL, NULL, &zErr);
    printf("program : Committing transaction\n");
    sqlite3_exec(db, "COMMIT", NULL, NULL, &zErr);
    printf("program : Starting transaction\n");
    sqlite3_exec(db, "BEGIN", NULL, NULL, &zErr);
    printf("program : Aborting transaction\n");
    sqlite3_exec(db, "ROLLBACK", NULL, NULL, &zErr);
    // Test table events
    printf("program : Creating table\n");
    sqlite3_exec(db, "create table foo(x int, y int, z int)", NULL, NULL, &zErr);
    printf("program : Inserting record\n");
    sqlite3_exec(db, "insert into foo values (1,2,3)", NULL, NULL, &zErr);
    printf("program : Selecting record (value for z should be NULL)\n");
    print_sql_result(db, "select * from foo");
    printf("program : Updating record (update of x should be denied)\n");
    sqlite3_exec(db, "update foo set x=4, y=5, z=6", NULL, NULL, &zErr);
    printf("program : Selecting record (notice x was not updated)\n");
    print_sql_result(db, "select * from foo");
    printf("program : Deleting record\n");
    sqlite3_exec(db, "delete from foo", NULL, NULL, &zErr);
    printf("program : Dropping table\n");
    sqlite3_exec(db, "drop table foo", NULL, NULL, &zErr);
```

Several things are going on here. The program selects all records in the table, one of which is

column z. We should see in the output that column z's value is NULL. All other fields should

contain data from the table. Next, the program attempts to update all fields, the most important

of which is column x. The update should succeed, but the value in column x should be

unchanged, as the authorizer denies it. This is confirmed on the following SELECT statement,

which shows that all columns were updated except for column x, which is unchanged. The

program then drops the foo table, which should issue a schema change notification from the previous filter.

```
    // Test ATTACH/DETACH
    // Connect to test2.db
    rc = sqlite3_open("test2.db", &db2);
    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db2));
        sqlite3_close(db2);
        exit(1);
     }
    // Drop table foo2 in test2 if exists
    sqlite3_exec(db2, "drop table foo2", NULL, NULL, &zErr);
    sqlite3_exec(db2, "create table foo2(x int, y int, z int)", NULL, NULL, &zErr);
    // Attach database test2.db to test.db
    printf("program : Attaching database test2.db\n");
    sqlite3_exec(db, "attach 'test2.db' as test2", NULL, NULL, &zErr);
    // Select record from test2.db foo2 in test.db
    printf("program : Selecting record from attached database test2.db\n");
    sqlite3_exec(db, "select * from foo2", NULL, NULL, &zErr);
    printf("program : Detaching table\n");
    sqlite3_exec(db, "detach test2", NULL, NULL, &zErr);
    /* -------------------------------------------------------------------------
    **  Cleanup
    ** -------------------------------------------------------------------------
    */
    sqlite3_close(db);
    sqlite3_close(db2);
    return 0;
}
```

线程

如第2章所述，SQLite支持线程。在多线程环境下使用SQLite时，有一些基本规则需要遵守。

共享缓冲区模式

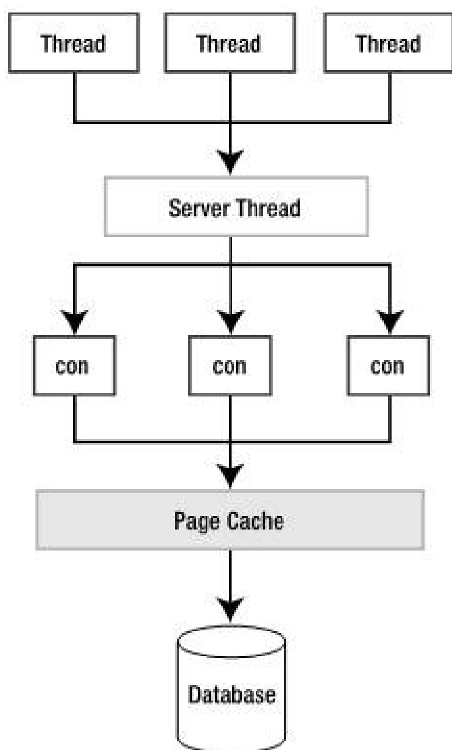


Figure 6-3. The shared cache model

共享缓冲区模式的目录是为了节省内在，SQLite中有几个函数是与线程和内存管理有关的。使用它们可以限制堆的尺寸或手工地发起内存清理。这些函数包括：

```
void sqlite3_soft_heap_limit(int N);
int sqlite3_release_memory(int N);
void sqlite3_thread_cleanup(void);
```

第7章 扩充C API

本章介绍SQLite的新技巧。前一章涉及一般的数据库操作，本章将开始创新。扩充API提供3种基本方法来扩展(或说定制)SQLite，包括：创建用户自定义函数、聚合和排序序列。

用户自定义函数是编写用于特定应用的SQL函数。一旦注册，就可以在SQL中被调用。

本章将涉及所有这3个用户定义的扩展工具及与之相关的API函数。你会看到，当与其它工具，如触发器和冲突解决等结合在一起时，用户定义的扩充API是强有力的，并能为SQLite创造非凡特色。

空注：本章内容对编程还是很有用的，但我对这部分内容只是略做浏览。关心这部分内容的兄弟还是得自己看原文。

API

用户自定义聚合、函数和排序法的生命同期是基于连接的。它们不存储在数据库中。有时你可能会把它们当成存储过程看待，而忘记了它们是在数据库之外的。它们存在于程序库(librarie)中，其生命周期严格地限制在你的程序之内。I

注册函数

步进函数

自定义函数和聚合的步进函数是一样的，可如下定义：

```
void fn(sqlite3_context* ctx, int nargs, sqlite3_value** values)
```

返回值

函数

返回值

一个完整的例子

Listing 7-2. The main Function

```
int main(int argc, char **argv)
{
    int rc;
    sqlite3 *db;
    const char* sql;
    sqlite3_open("test.db", &db);
    sqlite3_create_function( db, "function", -1, SQLITE_UTF8, NULL,
                             function, NULL, NULL);
    /* Turn on SQL logging */
    //log_sql(db, 1);
    /* Call function with one text argument. */
    execute(db, "select function(1)");
    /* Call function with several arguments of various types. */
    execute(db, "select function(1, 2.71828)");
    /* Call function with variable arguments, the first argument’s value
    ** being 'fail'. This will trigger the function to call
    ** sqlite3_result_error(). */
    execute(db, "select function('fail', 1, 2.71828, 'three', X'0004', NULL)");
    /* Done */
    sqlite3_close(db);
    return 0;
}
```

Listing 7-3. A Vanilla User-Defined Function

```
void function(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    int i; const char *msg;
    fprintf(stdout, "function() : Called with %i arguments\n", nargs);
    for(i=0; i < nargs; i++) {
        fprintf( stdout, "    arg %i: value=%-7s type=%i\n", i,
                 sqlite3_value_text(values[i]),
                 sqlite3_value_type(values[i]));
    }
    if(strcmp((const char *)sqlite3_value_text(values[0]), "fail") == 0) {
        msg = "function() : Failing because you told me to.";
        sqlite3_result_error(ctx, msg, strlen(msg));
        fprintf(stdout, "\n");
        return;
    }
    fprintf(stdout, "\n");
    sqlite3_result_int(ctx, 0);
}
```

一个实际的应用程序

聚合

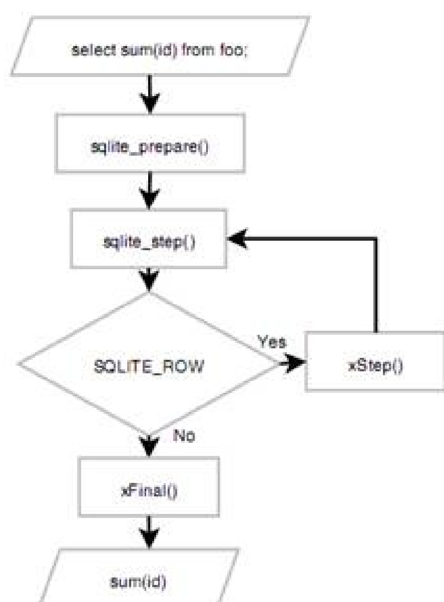


Figure 7-1. Query processing with aggregates

一个实际的例子

Listing 7-9. The sum_int() Test Program

```
int main(int argc, char **argv)
{
    int rc;
    sqlite3 *db;
    char *sql;
    rc = sqlite3_open("test.db", &db);
    if(rc) {
        print_error(db, "Can't open database");
        exit(1);
    }
    /* Create aggregate table, add records. */
    setup(db);
    /* Register aggregate. */
    fprintf(stdout, "Registering aggregate sum_int()\n");
    log_sql(db, 1);
    sqlite3_create_function( db, "sum_int", 1, SQLITE_UTF8, db,
                             NULL, step, finalize);
    /* Test it. */
    fprintf(stdout, "\nRunning query: \n");
    sql = "select sum_int(id) from aggregate";
    print_sql_result(db, sql);
    /* Done. */
    sqlite3_close(db);
    return 0;
}
```

步进函数

The step() function is shown in Listing 7-10.

Listing 7-10. The sum_int() Step Function

```
void step(sqlite3_context* ctx, int ncols, sqlite3_value** values)
{
    sum* s;
    int x;
    s = (sum*)sqlite3_aggregate_context(ctx, sizeof(sum));
    if(sqlite3_aggregate_count(ctx) == 1) {
        s->x = 0;
    }
    x = sqlite3_value_int(values[0]);;
    s->x += x;
    fprintf(stdout, "step()     : value=%i, total=%i\n", x, s->x);
}
The value sum is a struct that is specific to this example and is defined as follows:
typedef struct {
    int x;
} sum;
```

聚合的Context

Finalize函数

Listing 7-11. The sum_int() Finalize Function

```
void finalize(sqlite3_context* ctx)
{
    sum* s;
    s = (sum*)sqlite3_aggregate_context(ctx, sizeof(sum));
    sqlite3_result_int(ctx, s->x);
    fprintf(stdout, "finalize() : total=%i\n\n", s->x);
}
```

排序法

排序法定义

排序法如何工作

标准的排序法类型

一个简单的例子

Compare函数

Listing 7-12. The Political Collation Function

```
int political_collation( void* data, int l1, const void* s1,
                                     int l2, const void* s2 )
{
    int value, opinion; struct tm* t; time_t rt;
    /* Get the unpolitical value */
       value = strcmp(s1,s2);
    /* Get the date and time */
    time(&rt);
    t = localtime(&rt);
    /* Form an opinion */
       switch(t->tm_wday) {
        case 0: /* Monday yes    */
            opinion = value;
            break;
        case 1: /* Tueday no     */
            opinion = -value;
            break;
        case 2: /* Wednesday bigger is better */
            opinion = l1 >= l2 ? -1:1;
            break;
        case 3: /* Thursday strongly no   */
            opinion = -100;
            break;
        case 4: /* Friday strongly yes    */
            opinion = 100;
            break;
        case 5: /* Saturday golf, everything's the same */
            opinion = 0;
            break;
        default: /* Sunday - Meet the Press, opinion changes
                    by the hour */
            opinion = 2*(int)sin(t->tm_hour*180);
    }
    /* Now change it on a whim */
    opinion = rand()-(RAND_MAX/2) > 0 ? -1:1;
    return opinion;
}
```

测试程序

Listing 7-13. The Political Collation Test Program

```
int main(int argc, char **argv)
{
    int rc;
    sqlite3 *db;
    char *sql;
    /* For forming more consistent political opinions. */
    srand((unsigned)time(NULL));
    rc = sqlite3_open("test.db", &db);
    if(rc) {
        print_error(db, "Can't open database");
        exit(1);
    }
    /* Create issues table, add records. */
    setup(db);
    /* Register collating sequence. */
    fprintf(stdout, "1. Register political collating sequence\n\n");
    sqlite3_create_collation( db, "POLITICAL",
                              SQLITE_UTF8, db,
                              political_collation );
    /* Turn SQL logging on. */
    log_sql(db, 1);
    /* Test default collation. */
    fprintf(stdout, "2. Select records using default collation.\n");
    sql = "select * from issues order by issue";
    print_sql_result(db, sql);
    /* Test Oracle collation. */
    fprintf(stdout, "\nSelect records using political collation. \n");
    sql = "select * from issues order by issue collate POLITICAL";
    print_sql_result(db, sql);
    /* Done. */
    sqlite3_close(db);
    return 0;
}
```

按需排序(Collation on Demand)

Listing 7-14. Collation Registration Function

```
void crf( void* data, sqlite3* db,
    int eTextRep, const char* cname)
{
    if(strcmp(collation_name, "POLITICAL") == 0) {
        /* Political collation has not been registered and is now needed */
        sqlite3_create_collation( db, "POLITICAL",
            SQLITE_UTF8, db,
            political_collation );
    } else {
        /* Punt: Use some default comparison function this collation. */
        sqlite3_create_collation( db, collation_name,
            SQLITE_UTF8, db,
            default_collation );
    }
}
```

一个实际的应用程序

比较函数

Listing 7-15. Oracle Date Collation Function

```
int oracle_date_collation( void* data,
                           int len1, const void* arg1,
                           int len2, const void* arg2 )
{
    int len;
    date d1;
    date d2;
    char zDate1[25];
    char zDate2[25];
    /* Copy date 1 */
    if(len1 > 24) {
        len = 24;
    } else {
        len = len1;
    }
    strncpy(&zDate1[0], arg1, len);
    zDate1[len] = '\0';
    /* Copy date 2 */
    if(len2 > 24) {
        len = 24;
    } else {
        len = len2;
    }
    strncpy(&zDate2[0], arg2, len);
    zDate2[len] = '\0';
    /* Convert dates to date struct */
    oracle_date_str_to_struct(zDate1, &d1);
    oracle_date_str_to_struct(zDate2, &d2);
    fprintf(stdout, "collate_fn() : date1=%s, date2=%s\n", zDate1, zDate2);
    /* Compare structs */
    if(d1.year < d2.year)
    {
        return -1;
    }
    else if(d1.year > d2.year)
    {
        return 1;
    }
    /* If this far, years are equal. */
    if(d1.month < d2.month)
    {
        return -1;
    }
    else if(d1.month > d2.month)
    {
        return 1;
    }
    /* If this far, months are equal. */
    if(d1.day < d2.day)
    {
        return -1;
    }
    else if(d1.day > d2.day)
    {
        return 1;
    }
    /* If this far, dates are equal. */
    return 0;
}
```

日期解析

Listing 7-16. The Oracle Date Parsing Function

```
int oracle_date_str_to_struct(const char* value, date* d)
{
    const char* date, *tmp;
    char *start, *end, zDay[3], zMonth[4], zYear[3];
    date = get_date(value);
    if(date == NULL) {
        fprintf(stderr, "Invalid date\n");
        return -1;
    }
    /* Find first '-' */
    start = strchr(date,'-');
    /* Find last '-' */
    end   = strchr(start+1,'-');
    /* Extract day part, convert to int*/
    strncpy(zDay, date,2);
    zDay[2] = '\0';
    d->day = atoi(zDay);
    /* Extract month part, convert to int*/
    strncpy(zMonth, start+1,3);
    zMonth[3] = 0;
    tmp = uppercase(zMonth);
    d->month = month_num(tmp);
    free((void*)tmp);
    /* Extract year part, convert to int*/
    strncpy(zYear, end+1,2);
    zYear[2] = '\0';
    d->year = atoi(zYear);
    free((void*)date);
    return 0;
}
```

Listing 7-17. The get_date() Function

```
#define ORACLE_DATE_REGEX "[0-9]{1,2}-[a-zA-Z]{3,3}-[0-9]{2,2}";
const char* get_date(const char* value)
{
    pcre *re;
    const char *error, *pattern;
    int erroffset;
    int ovector[3];
    int value_length;
    int rc, substring_length;
    char* result, *substring_start;
    pattern = ORACLE_DATE_REGEX;
    re = pcre_compile(
        pattern,              /* the pattern */
        0,                    /* default options */
        &error,               /* for error message */
        &erroffset,           /* for error offset */
        NULL);                /* use default character tables */
    /* Compilation failed */
    if (re == NULL) {
        return NULL;
    }
    value_length = (int)strlen(value);
    rc = pcre_exec(
        re,           /* the compiled pattern */
        NULL,         /* no extra data - we didn't study the pattern */
        value,        /* the value string */
        value_length, /* the length of the value */
        0,            /* start at offset 0 in the value */
        0,            /* default options */
        ovector,      /* output vector for substring information */
        3);           /* number of elements in the output vector */
    if (rc < 0) {
        /* Match error */
        return NULL;
    }
    /* Match succeded */
    substring_start = (char*)value + ovector[0];
    substring_length = ovector[1] - ovector[0];
    //printf("%.*s\n", substring_length, substring_start);
    result = malloc(substring_length+1);
    strncpy(result, substring_start, substring_length);
    result[substring_length] = '\0';
    return result;
}
```

测试程序

All three of the above functions work together to collate Oracle dates in chronological order. Our example program is shown in Listing 7-18.

Listing 7-18. The Oracle Collation Test Program

```
int main(int argc, char **argv)
{
    int rc;
    sqlite3 *db;
    char *sql;
    rc = sqlite3_open("test.db", &db);
    if(rc) {
        print_error(db, "Can't open database");
        exit(1);
    }
    /* Install oracle related date functions. */
    install_date_functions(db);
    /* Register collating sequence. */
    fprintf(stdout, "Registering collation sequence oracle_date\n");
    sqlite3_create_collation( db, "oracle_date",
                              SQLITE_UTF8, db,
                              oracle_date_collation );
    /* Create dates table, add records. */
    setup(db);
    /* Install date */
    install_date_triggers(db);
    /* Turn SQL logging on. */
    log_sql(db, 1);
    /* Test default collation. */
    fprintf(stdout, "Select records. Use default collation.\n");
    sql = "select * from dates order by date";
    print_sql_result(db, sql);
    /* Test Oracle collation. */
    fprintf(stdout, "\nSelect records. Use Oracle data collation. \n");
    sql = "select * from dates order by date collate oracle_date";
    print_sql_result(db, sql);
    /* Get ISO Date from Oracle date. */
    fprintf(stdout, "\nConvert Oracle date to ISO format.\n");
    sql = "select iso_from_oradate('01-APR-05') as 'ISO Date'";
    print_sql_result(db, sql);
    /* Validate Oracle date. */
    fprintf(stdout, "\nValidate Oracle format. Should fail.\n");
    sql = "select validate_oradate('01-NOT-2005')";
    execute(db, sql);
    /* Test Oracle date triggers. */
    fprintf(stdout, "\nTest Oracle insert trigger -- should fail.\n");
    sql = "insert into dates (date) values ('01-NOT-2005')";
    execute(db, sql);
    fprintf(stdout, "\nTest Oracle update trigger -- should succeed.\n");
    sql = "update dates set date='01-JAN-2005'";
    execute(db, sql);
    print_sql_result(db, "select * from dates");
    /* Done. */
    sqlite3_close(db);
    return 0;
}
```

运行结果

略。

第8章 语言扩展

SQLite本身是用C语言编写的，它有自己的C API。但是，开源社区中提供了多种SQLite的扩展，使你在以在其它的编程语言或程序库中存取SQLite数据库，如Per、Python、Ruby、Java、Qt和ODBC。在很多情况下，每种编程语言可以有几个扩展可供选择，这些扩展由不同的人为不同的需求而编写。

空注：我只在其它地方看了Delphi的扩展，本章基本没看。

第9章 SQLite内核

本章是SQLite各主要子系统的一个概览。它的灵感来自一次会议上Richard Hipp对SQLite所做的介绍。即使你没有看过SQLite的源代码，你也会发现这些内容是如此的有趣。即使SQLite还在发展，但本章所介绍的概念一时不会改变。

现在，你应该已经熟悉SQLite的主要组件了。第1章有一个概述，第5章介绍了B-tree和pager，这些概念本章就不再介绍了。本章会从虚拟机入手，它是SQLite的心脏；然后是存储层；最后是编译器，它可能是系统中最复杂的部分。

虚拟数据库引擎(VDBE)

VDBE是SQLite的核心，它的上层模块和下层模块本质上都是为它服务的，它的实现位于vbde.c、vdbe.h、vdbeapi.c、vdbeInt.h和vdbemem.c等几个文件中。如第5章所述，一个语句(statement)会编译为一个完整的VDBE程序，执行一条单独的SQL命令。它通过底层的基础设施B-tree执行由编译器(Compiler)生成的字节代码，这种字节代码程序语言是为了进行查询、读取和修改数据库而专门设计的。

字节代码在内存中被封装成sqlite3_stmt对象(内部叫做Vdbe，见vdbeInt.h)，Vdbe(或者说statement)包含执行程序所需要的一切，包括：

- VDBE程序
- 程序计数器
- 结果字段的名称和类型
- 参数的绑定值
- 运行栈和固定数量的编号的内在单元
- 其它的运行时状态信息，如B-tree游标

VDBE是一个虚拟机，它的字节代码指令和汇编程序十分类似，每一条指令由操作码和三个操作数构成：<opcode, P1, P2, P3>。Opcode为一定功能的操作码，为了理解，可以看成一个函数。p1是32位的有符号整数，p2是31位的无符号整数，它通常是跳转(jump)指令的目标地址(destination)，当然还有其它用途；p3为一个以null结尾的字符串或者其它结构体的指针。目前SQLite中有128个操作码。和C API不同的是，VDBE操作码经常变化，所以不应该用字节码编写自己的程序。

下面的几个C API直接和VDBE交互：

- sqlite3_bind_xxx() functions

- sqlite3_step()
- sqlite3_reset()
- sqlite3_column_xxx() functions
- sqlite3_finalize()

一般情况下，所有的API都是用来执行一个查询并在VDBE相关的结果集中步进操作。它们有一个共同点：都以一个语句句柄做参数。这是因为它们都需要句柄中的VDBE代码或相关资源来完成任务。注意：sqlite3_prepare()工作于开始阶段，用于产生VDBE代码，它不参与执行。

所有SQL命令的VDBE程序都可以通过EXPLAIN命令得到，如：

```
sqlite> .m col
sqlite> .h on
sqlite> .w 4 15 3 3 15
sqlite> explain select * from episodes;
addr         opcode                 p1     p2     p3
0             Goto                   0      12
1             Integer                0      0
2             OpenRead            0      2      # episodes
3             SetNumColumns   0      3
4             Rewind               0      10
5             Recno                  0      0
6             Column               0      1
7             Column               0      2
8             Callback               3      0
9             Next                    0      5
10           Close                  0      0
11           Halt                   0      0
12           Transaction           0      0
13           VerifyCookie         0      10
14           Goto                   0      1
15           Noop                   0      0
```

上面使用了4条命令，前面的命令用于调试和格式化。另外，我在编译SQLite时使用了SQLITE_DEBUG选项，这个选择可以提供运行栈更多的信息，比如包含在p3里面的表名。

空注：当前版本的SQLite(3.6.18)确实有较大变化，现在执行EXPLAIN命令的结果如下。

```
addr  opcode           p1   p2   p3     p4          p5    comment
----  ---------------  ---  ---  -----  ----------  ----  ----------
0     Trace            0    0    0                  00
1     Goto             0    11   0                  00
2     OpenRead         0    2    0      3           00
3     Rewind           0    9    0                  00
4     Rowid            0    1    0                  00
5     Column           0    1    2                  00
6     Column           0    2    3                  00
7     ResultRow        1    3    0                  00
8     Next             0    4    0                  01
9     Close            0    0    0                  00
10    Halt             0    0    0                  00
11    Transaction      0    0    0                  00
12    VerifyCookie     0    40   0                  00
13    TableLock        0    2    0      episodes    00
14    Goto             0    2    0                  00
```

空注：有关VDBE的最详细参考在vbde.c中，也可以参考SQLite网站提供的文档http://www.sqlite.org/opcode.html。

空注：后面的内容还按原文翻译。

栈(Stack)

一个VDBE程序通常由几个完成特定任务的段(section)构成，每一个段中都有一些操作栈的指令。这么做是因为不同的指令有不同数量的参数，有些指令只有一个参数；有些指令没有参数；有些指令有好几个参数，这时三个操作数就不够了。

考虑到这些情况，指令采用栈来传递参数。而这些指令本身不会做这些工作，所以在它们之前需要其它一些指令的帮助，以取得需要的参数。VDBE把计算的中间结果保存到内存单元(memory cell)中，其实堆栈和内存单元都基于Mem结构(见vdbeInt.h)。

程序体

让我们来看前面打开episodes表的例子。它的第一个段主要包括指令1~3。

第一条指令(Integer)是为第二条指令作准备的，也就是把第二条指令执行需要的参数压入堆栈，OpenRead从堆栈中取出参数值然后执行。

SQLite可以通过ATTACH命令在一个连接中打开多个数据库文件，每当SQLite打开一个数据库，它就为之赋一个索引号(index)，主数据库的索引为0，附加的第一个数据库为1，依次类推。Integer指令将数据库索引的值压入栈(本例为0，代表主数据库)，而OpenRead从中取出值，并决定操作哪个数据库。它用P2来确定需要打开表的根页(root page)。然后它打开一个指定数据库中指定表的B-tree游标。所有这些在VDBE代码文档中都有解释，例如，OpenRead命令在SQLite文档中的解释如下：

为数据库表打开一个只读游标，这个表的根页在数据库文件的P2处。数据库文件由栈顶的一个整数指定。0表示主数据库，1表示用于存放临时表的数据库。新打开游标的标识符在P1中。P1的值不必是相邻的，但应该是一个小整数。如果其值为负，表示错误。If P2==0 then take the root page number from off of the stack.

只要有游标打开，就会有一个读锁加载到数据库上。如果数据库本来是未加锁的，此命令的部分工作包括获得一个读锁。读锁允许其它进程读数据库，但是禁止任何进程改数据库。读锁在所有游标都关闭时释放。如果此指令在申请读锁时失败，程序结束并返回SQLITE_BUSY错误码。

P3的值是指向一个结构的指针，该结构定义索引的内容和排序序列的关键信息。当不指向索引时，P3的内容为空。

这个关于OpenRead的文档与其它指令的文档一样，可以直接在源程序文件中找到，特别是vdbe.c中。

最终，SetNumColumns指令设置游标需要处理的列的数量，这是由所要处理的表包含的列数决定的。P1为游标的索引(这里为0，是刚刚打开的游标的索引号)。P2为列的数目，episodes表有三列。

继续本例，Rewind指令将游标设置到表的开始，它会检查表是否为空(“空”即没有记录)。如果没有记录，它会导致指令指针跳转到P2指定的指令处。此处P2为10，即Close指令。一旦Rewind设置游标，接下来就会执行下一段(指令5~9)的几条指令。它们的主要功能是遍历结果集，Recno把由游标P1指定的记录的关键字段值压入堆栈。Column指令从由P1指定的游标，P2指定的列取值。5,6,7三条指令分别把id(primary key)、season和name字段(游标0所指明的表episodes的全部3个字段)的值压入栈。接下来，Callback指令从栈中取出三个值(由P1指定)，然后形成一个记录数组，存储在内存单元 (memory cell) 中。然后，Callback会挂起VDBE的执行，把控制权交给sqlite3_step()，该函数将返回SQLITE_ROW。

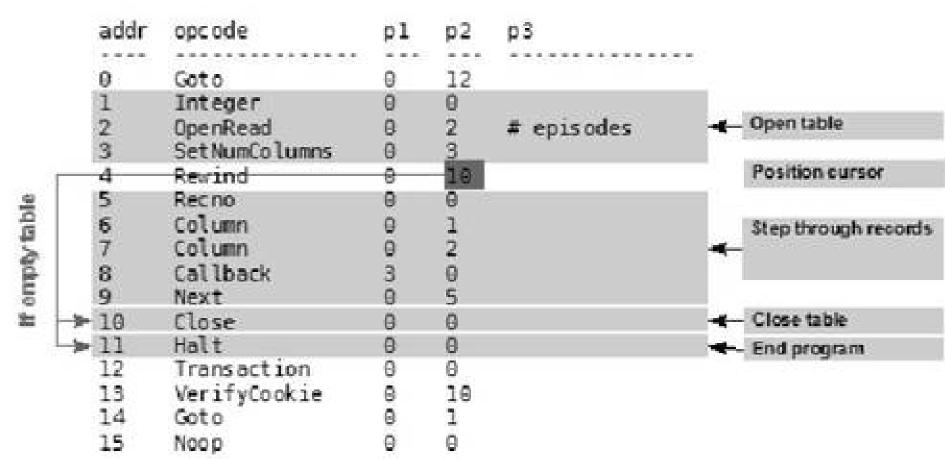


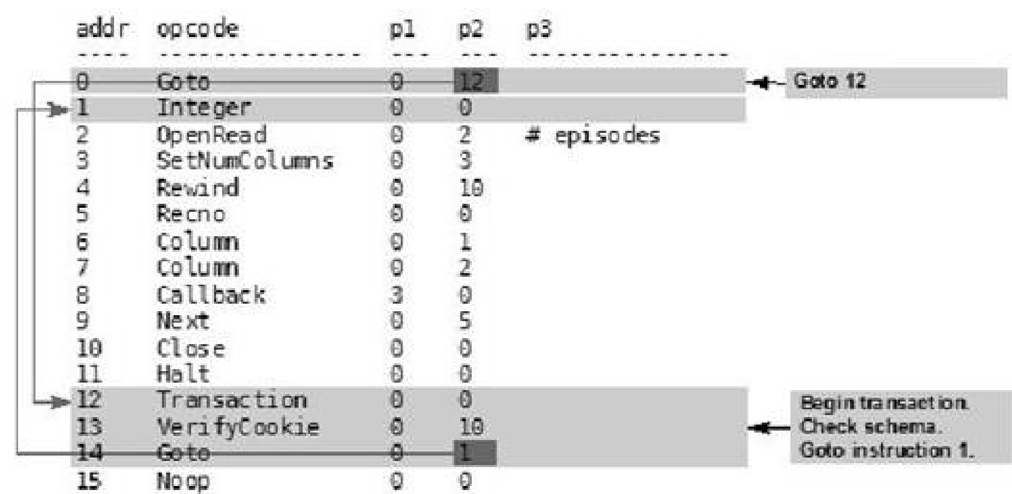
图9-1 VDBE的步骤：Open和Read

一旦VDBE创建了记录结构(该结构同样关联于语句(statement)句柄)，程序就可以通过sqlite3_column_xxx() 函数从记录结构内取出字段值。当下次调用sqlite3_step()时，指令指针会指向Next指令。Next指令会把游标移向表的下一行，如果有其它的记录，它会跳到由P2指定的指令，在这里为指令5(Recno指令)，创建一个新的记录结构，进入下一次循环。如果已经没有其它记录可读，Next不跳转，而是执行下一条指令，这里是Close指令。Close指令会关闭游标，然后执行Halt指令，结束VDBE程序，并且sqlite3_step()函数会返回SQLITE_DONE。

程序开始与停止

前面介绍了程序的核心部分，现在来看看其余的指令，这些指令与启动和初始化有关，见图9-2。第一条指令是Goto指令，它是一条跳转指令，跳到P2处，本例中是跳到第12条指令。

指令12是Transaction，它开始一个新的事务；然后执行下一条指令VerifyCookie，它的主要功能是确定VDBE程序编译后，数据库schema是否改变(即是否进行过更新操作)。这在SQLite中是一个很重要的概念，在SQL被sqlite3_prepare()编译成VDBE代码至程序调用sqlite3_step()执行字节码的这段时间内，另一个SQL命令可能会改变数据库模式(比如ALTER TABLE、DROP TABLE或CREATE TABLE)。一旦发生这种情况，schema版本就会改变，之前编译的语句(statement)就会变得无效。当前的数据库schema信息记录在数据库文件的根页中。类似地，每个语句都有一份用于比较的在编译时刻该模式的备份，VerifyCookie的功能就是检查它们是否匹配，如果不匹配，就要采取适当的措施。



addr	opcode	p1	p2	p3
0	Goto	0	12	
1	Integer	0	0	
2	OpenRead	0	2	# episodes
3	SetNumColumns	0	3	
4	Rewind	0	10	
5	Recno	0	0	
6	Column	0	1	
7	Column	0	2	
8	Callback	3	0	
9	Next	0	5	
10	Close	0	0	
11	Halt	0	0	
12	Transaction	0	0	
13	VerifyCookie	0	10	
14	Goto	0	1	
15	Noop	0	0	

图9-2 VDBE的步骤：程序开始

语句的版本号由VerifyCookie的P2参数指定，将它与磁盘上的数据库schema版本号进行比较。如果schema没有改变，两个版本号应该一致。如果不一致，则VDBE程序失效。在此情况下，VerifyCookie将会终止程序并返回SQLITE_SCHEMA错误。在此情况下，应用程序需要重新编译SQL语句，基于新的schema版本生成新的VDBE程序。

如果两者匹配，会执行下一条指令Goto；它会跳到程序的主要部分，即第一条指令，打开表读取记录。这里有两点值得注意：

(1)Transaction指令本身不会获取锁。它的功能相当于BEGIN，而共享锁实际是由OpenRead指令获取的。当事务关闭时释放锁，由Halt指令完成，它会进行扫尾工作。

(2) 语句对象(VDBE程序)所需的存储空间在程序执行前就已经确定。这缘于两个重要事实：首先，栈的深度不会比指令的数目还多。其次，内存单元(memory cell)的数量永远不会多于指令的数量(通常少得多)。在执行VDBE程序之前，SQLite可以计算出分配资源所需要的内存。

指令的类型

VDBE同时只会执行一条指令。每条指令都完成一项简单的任务，而且通常和该指令前面、后面的指令有关。大体上来说，指令可分为三类：

(1)处理值：这些指令通常完成算术运算，比如加、减和除；逻辑运算，比如与和或；还有字符串操作。

(2)数据管理：这些指令操作在内存和磁盘上的数据。内存指令进行栈操作或者在内存单元之间传递数据。磁盘操作指令控制B-tree和pager打开或操作游标，开始或结束事务，等等。

(3)流程控制：控制指令主要是有条件地或无条件地移动指令指针。

一旦熟悉了指令集，就不难明白VDBE程序是如何工作的。至少你可以了解如何使用栈来为后面指令的执行做准备。

B-Tree和Pager模型

B-tree使VDBE执行查找、插入和删除的效率达到O(logN)，以及在O(1)的效率下双向遍历结果集。它是自平衡的，可自动地执行碎片整理和空间回收。B-tree本身不会直接读写磁盘，它仅仅维护着页(page)之间的关系。当B-tree需要页或者修改页时，它就会调用pager。当修改页时，pager保证原始页首先写入日志文件。当它完成写操作时，pager根据事务状态决定如何做。

数据库文件格式

数据库中所有的页从1开始顺序编号。一个数据库由多个多重B-tree构成——B-tree用于每一个表和索引。每个表和索引的第1个页(地址)称为根页。所有表和索引的根页都存储在sqlite_master表中。

数据库中第一个页(page 1)有点特殊，page 1的前100个字节是一个对数据库文件进行描述的特殊文件头。它包括库的版本、格式的版本、页大小、编码等所有创建数据库时设置的永久性参数。有关这个特殊文件头的文档在btree.c中，page 1也是sqlite_master表的根页。

页重用及回收

SQLite利用一个空闲页链表(free list)完成页的循环使用。当一个页的所有记录都被删除时，就被插入到该链表。当有新信息需要进入数据库时，临近的空闲页先被选中，当没有空闲页时，才创建新的页(会增加文件的大小)。当运行VACUUM命令时，会清空空闲页链表，所以数据库会缩小。本质上它是在新的文件中重新建立数据库，而所正使用的页都被拷贝过去，而空闲页链表不拷，结果就是一个新的，变小了的数据库。当数据库的autovacuum开启时，SQLite不会使用空闲页链表，而且在每一次事务提交时自动压缩数据库。

B-Tree记录

B-tree中的页由B-tree记录组成，也叫做payload(有效载荷)。每一个B-tree记录(或payload)有两个域：关键字域(key field)和数据域(data field)。关键字域就是ROWID的值，也就是每个数据库表都会提供的关键字的值。从B-tree的角度，数据域可以是任何无结构的数据。数据库的记录就保存在这些数据域中。B-tree的任务就是排序和遍历，这仅需要关键字段。Payload的大小是不定的，这与内部的关键字和数据域有关。平均情况下，每个页一般包含多个payload，当然也可能一个payload占用几个页(当一个payload太大不能存在一个页内)。

B+树

B-tree按关键字的顺序存储，在一个B-tree中所有的关键字必须唯一(这一点自动地由ROWID主键字段保证)。表采用B+tree，B+tree的内部结点不包含表数据(数据库记录)。图9-3是一个表的B+tree的示例：

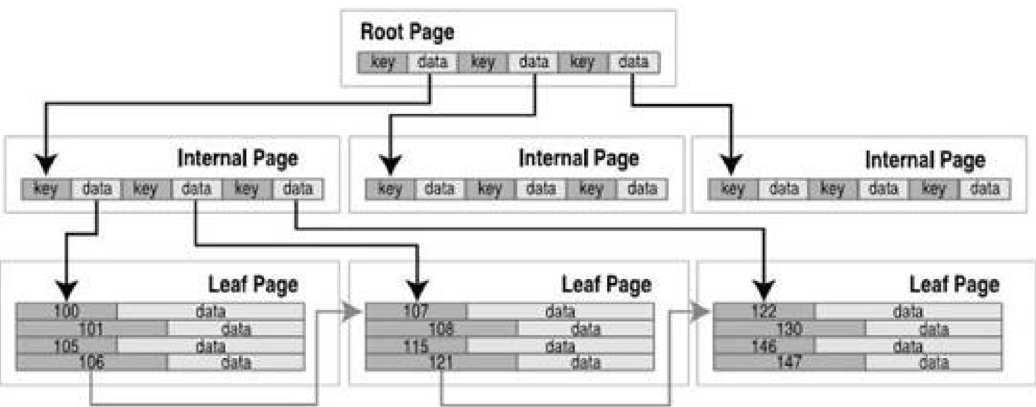


图9-3 B+tree的组织(表)

B+tree中的根页(root page)和内部页(internal page)都是用来导航的，这些页的数据域都是指向下级页的指针，仅仅包含关键字。所有的数据库记录都存储在叶子页(leaf page)内。在叶节点一级，记录和页都是按照关键字的顺序排列的，这使B-tree游标只使用页结点就能正向和反向(水平地)遍历记录，并使遍历的效率(时间复杂度)可能达到O(1)。

记录和字段

数据库记录位于叶子页的数据域，由VDBE管理(前面在介绍Callback命令时介绍过)。数据库记录以二进制的形式存储，但有一定的数据格式，这种格式描述了记录中的所有字段。记录格式是连续的字节流，其组成包括一个逻辑头(logical header)和一个数据区(data segment)，逻辑头包括“头大小(可变长的64位整数)”和一个数据类型(也是可变长的64位整数)数组，数据类型用来描述存储在数据区的字段的类型，如图9-4所示。64位整数用Huffman编码实现。

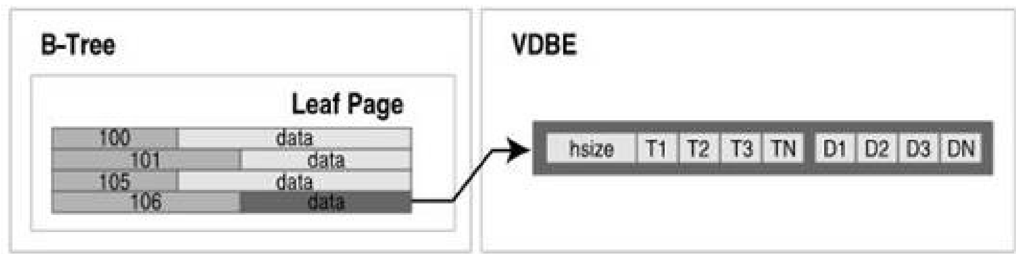


图9-4 记录结构

类型入口的数量与字段数量相等。类型数组与字段数组的元素按下标相对应。一个类型入口表明它对应字段的数据类型和宽度。类型入口的可能取值及其含义在表9-1中列出。

表9-1 字段类型值

类型值	含义	数据宽度
0	NULL	0
N in 1..4	有符号整数	N
5	有符号整数	6
6	有符号整数	8
7	IEEE符点数	8
8-11	未使用	N/A
N>12的偶数	BLOB	(N-12)/2
N>13的奇数	TEXT	(N-13)/2

例如，取episodes表的第1条记录：

```
sqlite> SELECT * FROM episodes ORDER BY id LIMIT 1;
id    season     name
0     1            Good News Bad News
```

这条记录的内部记录格式如图9-5所示。



表9-5 episodes表的第1条记录

记录头长4字节。头的大小反映头内各要素都是单字节编码。第一个类型，对应id字段，是一个1字节有符号整数。第二个类型，对应season字段，也是一个1字节有符号整数。Name字段的类型入口是一个大于13的奇数，表示它是一个text值，该值占(49-13)/2=18个字节。通过这些信息，VDBE可以解析记录的数据段并取出独立的字段值。

层次数据组织

SQLite的层次数据组织模型如图9-6所示。在模型中，每层处理特定的数据单元。从下向上，数据越来越结构化；从上往下，数据越来越无序。C-API处理字段值，VDBE处理记录，B-tree处理键值的数据，pager处理页，OS接口处理二进制的数据和原始存储器。

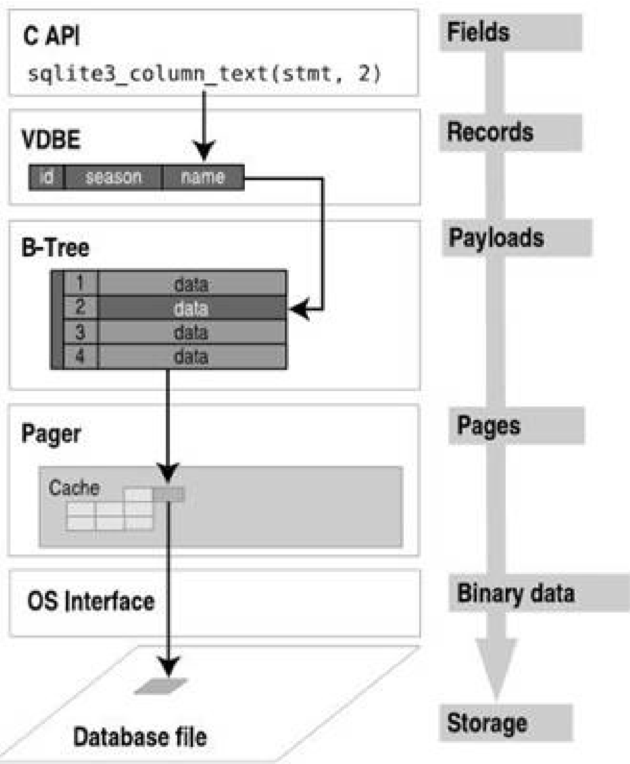


图9-6 模型和各层所对应的数据

Each module takes part in managing its own specific portion of the data in the database, and relies on the layer below it to supply it with a more crude form from which to extract its respective pieces.

溢出页

如前所述，B-tree记录具有可变的大小，而页的大小是固定的。这就有可能一个B-tree记录比一个单独的页还要大。这时，超大的B-tree记录就溢出到由溢出页组成的链表上，如图9-7所示。

在图中，第4个页太大，B-tree模块就创建一个溢出页来容纳它。如果一个溢出页还不够，就再链接第2个。这实际上也是二进制大对象的处理方法。请记住：当你使用大的BLOB时，它实际上是存储在页链表中的。如果BLOB实在太大，链表就会很长，操作就会很低效。这种情况下，将BLOB存储在一个外部文件中而在数据库中只保存其文件名也许更好一些。

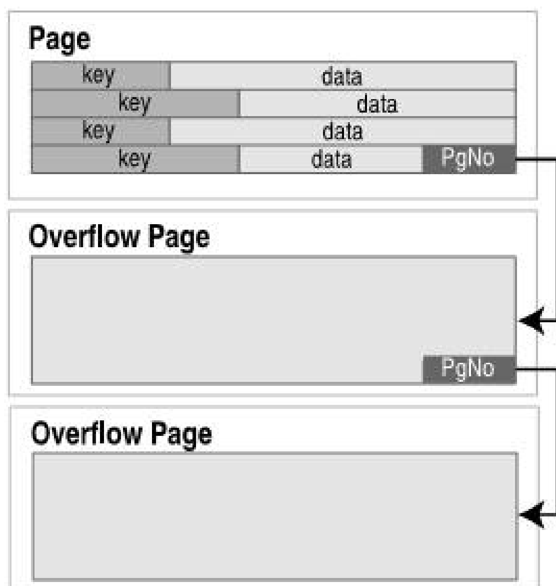


图9-7 溢出页

B-Tree API

B-Tree模块有它自己的API，它可以独立于C API使用。也就是说，如果你愿意，你可以把它当作一个独立的运行库来使用，或在SQLite中直接存取库表。SQLite B-tree模块的另一个好处就是它本身支持事务。由pager处理的事务、锁和日志都是为B-tree服务的。根据功能，可将B-Tree API分为以下几类：

访问和事务函数

包括：

- sqlite3BtreeOpen: 打开一个新的数据库文件，返回一个B-tree对象。
- sqlite3BtreeClose: 关闭一个数据库。
- sqlite3BtreeBeginTrans: 开始一个新的事务。
- sqlite3BtreeCommit: 提交当前事务。
- sqlite3BtreeRollback: 回卷当前事务。
- sqlite3BtreeBeginStmt: 开始一个statement事务。
- sqlite3BtreeCommitStmt: 提交一个statement事务。
- sqlite3BtreeRollbackStmt: 回卷一个statement事务。

表函数

包括：

- sqlite3BtreeCreateTable: 在数据库文件中创建一个新的、空的B-tree。其参数决定是采用表格式(B+tree)还是索引格式(B-tree)。
- sqlite3BtreeDropTable: 从数据库中删除一个B-tree。
- sqlite3BtreeClearTable: 从B-tree中删除所有数据，但保持B-tree的结构。

游标函数

包括：

- sqlite3BtreeCursor: Creates a new cursor pointing to a particular B-tree. Cursors can be either a read cursor or a write cursor. Read and write cursors may not exist in the same B-tree at the same time.
- sqlite3BtreeCloseCursor: Closes the B-tree cursor.
- sqlite3BtreeFirst: Moves the cursor to the first element in a B-tree.
- sqlite3BtreeLast: Moves the cursor to the last element in a B-tree.
- sqlite3BtreeNext: Moves the cursor to the next element after the one it is currently pointing to.
- sqlite3BtreePrevious: Moves the cursor to the previous element before the one it is currently pointing to.
- sqlite3BtreeMoveto: Moves the cursor to an element that matches the key value passed in as a parameter. If there is no match, leaves the cursor pointing to an element that would be on either side of the matching element, had it existed.

记录函数

包括：

- sqlite3BtreeDelete: Deletes the record that the cursor is pointing to.
- sqlite3BtreeInsert: Inserts a new element in the appropriate place of the B-tree.
- sqlite3BtreeKeySize: Returns the number of bytes in the key of the record that the cursor is pointing to.
- sqlite3BtreeKey: Returns the key of the record the cursor is currently pointing to.
- sqlite3BtreeDataSize: Returns the number of bytes in the data record that the cursor is currently pointing to.

- sqlite3BtreeData: Returns the data in the record the cursor is currently pointing to.

配置函数

包括：

- sqlite3BtreeSetCacheSize: Controls the page cache size as well as the synchronous writes (as defined in the synchronous pragma).
- sqlite3BtreeSetSafetyLevel: Changes the way data is synced to disk in order to increase or decrease how well the database resists damage due to OS crashes and power failures. Level 1 is the same as asynchronous (no syncs() occur and there is a high probability of damage). This is the equivalent to pragma synchronous=OFF. Level 2 is the default. There is a very low but non-zero probability of damage. This is the equivalent to pragma synchronous=NORMAL. Level 3 reduces the probability of damage to near zero but with a write performance reduction. This is the equivalent to pragma synchronous=FULL.
- sqlite3BtreeSetPageSize: Sets the database page size.
- sqlite3BtreeGetPageSize: Returns the database page size.
- sqlite3BtreeSetAutoVacuum: Sets the autovacuum property of the database.
- sqlite3BtreeGetAutoVacuum: Returns whether the database uses autovacuum.
- sqlite3BtreeSetBusyHandler: Sets the busy handler.

还有其它的函数，所有这些函数在btree.h和btree.c中都有很完备的文档，但上面列出的函数可以使你建立一个总体印象。

编译器

前面已经介绍了VDBE以下直到OS层的各层次。下面介绍VDBE程序是怎么来的。编译器的输入是一个单独的SQL命令，输出是最终经过优化的VDBE程序，这些工作在3个阶段上完成：分词器(Tokenizer)、分析器(Parser)和代码生成器(Code Generator)。

分词器(Tokenizer)

编译的第一步是对SQL命令分词。分词器将一个命令分解成一串单独的词汇(token)。词可以是有特定含义的一个字符或一个字符序列。每个词都有其关联的词类(token class)，词类是一个数字标识，表明这个词是什么。例如左括号的词类是TK_LP，保留字SELECT的词类是TK_SELECT。所有词类在parse.h中定义。例如下面的SQL语句：

```
SELECT rowid FROM foo where name='bar' LIMIT 1 ORDER BY rowid;
```

经分词器处理之后的部分结果在表9-2中给出。

表9-2 一个分词后SELECT语句

文本	词类	动作
SELECT	TK_SELECT	发给分析器
" "	TK_SPACE	丢弃
Rowid	TK_ID	发给分析器
" "	TK_SPACE	丢弃
FROM	TK_FROM	发给分析器
" "	TK_SPACE	丢弃
foo	TK_ID	发给分析器
" "	TK_SPACE	丢弃
WHERE	TK_WHERE	发给分析器
" "	TK_SPACE	丢弃
name	TK_ID	发给分析器
=	TK_EQ	发给分析器
…		

总之，分词器按照SQL的词法定义把它切分为一个一个的词，并传递给分析器(Parser)进行语法分析(忽略空格)。

保留字

分词器是手工编写的(hand-coded)，主要在Tokenize.c中实现。因为是手工代码，不是用自动生成的代码来对SQL保留字分类。保留字在keywordhash.h文件中定义。这个文件是一个最优化的、将所有SQL保留字压缩到可能最小的缓冲区，方法是公共的字符序列重叠存放。SQLite使用指明了每个保留字偏移量和大小的数组来识别保留字入口。这种方法是一种空间优化的方法，有利于内嵌式的应用程序。一个生成了的缓冲区的例子如下：

```
static int keywordCode(const char *z, int n){
static const char zText[537] =
"ABORTABLEFTEMPORARYADDATABASELECTHENDEFAULTRANSACTIONATURALTER"
"AISEACHECKEYAFTEREFERENCESCAPELSEXCEPTRIGGEREGEXPLAINITIALLYANALYZE"
"XCLUSIVEXISTSTATEMENTANDEFERRABLEATTACHAVINGLOBEFOREIGNOREINDEX"
"AUTOINCREMENTBEGINNERENAMEBETWEENOTNULLIKEBYCASCADEFERREDELETE"
"CASECASTCOLLATECOLUMNCOMMITCONFLICTCONSTRAINTERSECTCREATECROSS"
"CURRENT_DATECURRENT_TIMESTAMPLANDESCDETACHDISTINCTDROPRAGMATCH"
"FAILIMITFROMFULLGROUPDATEIFIMMEDIATEINSERTINSTEADINTOFFSETISNULL"
"JOINORDEREPLACEOUTERESTRICTPRIMARYQUERYRIGHTROLLBACKROWHENUNION"
"UNIQUEUSINGVACUUMVALUESVIEWHERE";
```

The keywordhash.h file includes a routine sqlite3KeywordCode(), which allows the tokenizer to quickly match the keyword with its appropriate token class with minimal space. So, the tokenizer first tries to match a token with what it knows, and failing that, it resorts to sqlite3KeywordCode(), which will return either a keyword token class or a generic TK_ID.

The tokenizer and parser work hand in hand, one token at a time. As the tokenizer resolves each token, it passes the token to the parser. The parser takes the tokens and builds a parse tree, which is a hierarchical representation of the statement.

分析器(Parser)

SQLite的语法分析器是用Lemon生成的(Lemon是一个开源的LALR(1)语法分析器的生成器，SQLite在使用时进行了定制)。该分析器用parse.c内定义的语法规则将一串词组织成层次结构的分析树(parse tree)。

The parse tree is primarily composed of expressions and lists of expressions. An expression itself is a recursive structure that can contain subexpressions under it. For example, the WHERE clause in a SELECT parse tree is represented by a single expression. The SELECT clause, on the other hand, is represented as a list of expressions; each expression is a column that will be returned in the result set. 例如，如下简单的SQL语句：

```
SELECT rowid, name, season FROM episodes WHERE rowid=1 LIMIT 1
```

可以组织成如图9-8所示的分析树。

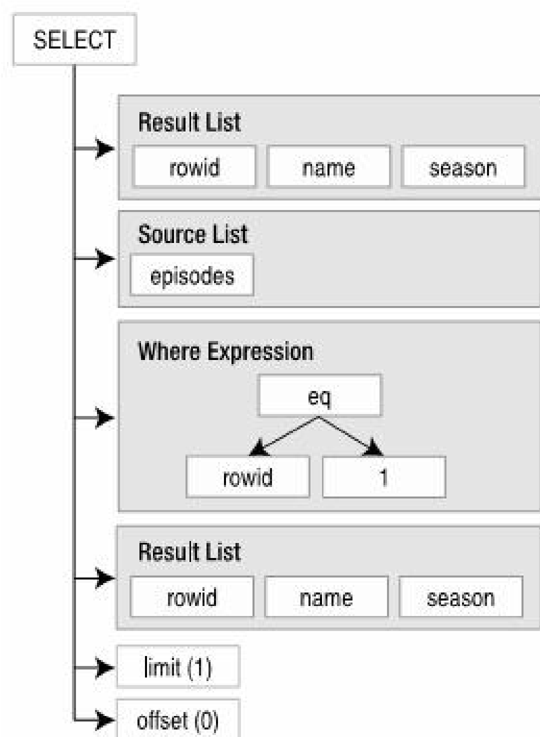


图9-8 简化了的分析树

一旦语句经过分词和分析，分析树作为一种结果会传送给代码生成器。

代码生成器(Code Generator)

代码生成器是SQLite中最庞大、最复杂的部分。与其它模块不同，代码生成器没有定义明确的接口，但它与分析器关系紧密。代码生成器由多个源文件组成，这些源文件大多针对特定的SQL操作。例如，生成SELECT语句的代码在select.c中，其它的源文件包括update.c、insert.c、delete.c和trigger.c等等。

代码生成器根据语法分析树生成VDBE程序。树的每一部分生成一个VDBE指令序列来完成特定的任务。The values for the operands are taken from the data structures associated with the parse tree. 例如，下面是一个读操作中打开表的代码的生成实现：

```
/* Generate code that will open a table for reading.*/
void sqlite3OpenTableForReading(
  Vdbe *v,        /* Generate code into this VDBE */
  int iCur,       /* The cursor number of the table */
  Table *pTab     /* The table to be opened */
){
  sqlite3VdbeAddOp(v, OP_Integer, pTab->iDb, 0);
  sqlite3VdbeAddOp(v, OP_OpenRead, iCur, pTab->tnum);
  VdbeComment((v, "# %s", pTab->zName));
  sqlite3VdbeAddOp(v, OP_SetNumColumns, iCur, pTab->nCol);
}
```

Sqlite3vdbeAddOp函数有三个参数：(1)VDBE实例(它将添加指令)，(2)操作码(一条指令)，(3)两个操作数。它增加一条指令到VDBE程序。上例中，sqlite3OpenTableForReading增加了3条指令，即图9-1中的指令1~3，功能是打开一个表的B-tree用于读。为方便起见，将图9-1中的指令序列重列于此：

```
sqlite> explain select * from episodes;
addr        opcode                p1    p2    p3
0            Goto                  0     12
1            Integer               0     0
2            OpenRead            0     2     # episodes
3            SetNumColumns   0     3
4            Rewind               0     10
5            Recno                 0     0
6            Column              0     1
7            Column              0     2
8            Callback              3     0
9            Next                   0     5
10          Close                  0     0
11          Halt                   0     0
12          Transaction          0     0
13          VerifyCookie        0     10
14          Goto                  0     1
15          Noop                  0     0
```

优化

代码生成器不仅负责生成代码，也负责进行查询优化。优化是代码生成的一部分，主要的实现位于where.c中。生成的WHERE子句通常被其它模块共享，比如select.c、update.c和delete.c。这些模块调用sqlite3WhereBegin()开始WHERE语句块的指令生成，然后将返回的代码加入到它们自己的VDBE代码中，最后调用sqlite3WhereEnd()，生成结束WHERE子句代码的VDBE指令。程序的一般结构如图9-9所示：

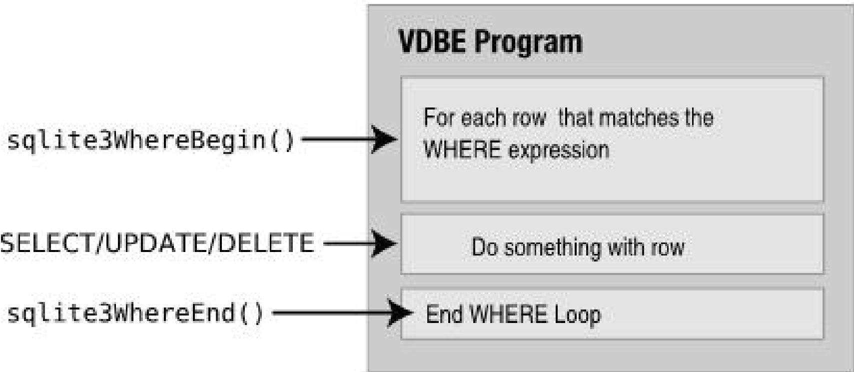


图9-9 WHERE子句的VDBE代码的生成 优化发生在sqlite3WhereBegin()阶段。 它在已完成工作的基础上，寻找可以使用的索引，寻找可以重写的表达式，等等。 为了能对优化先有一个感觉， 我们从一个不含WHERE子句的简单SELECT语句开始，如图9-10所示：

```
sqlite> explain SELECT name FROM episodes;
addr  opcode          p1   p2   p3
----  --------------  ---  ---  --------------
0     Goto            0    10
1     Integer         0    0
2     OpenRead        0    2    # episodes
3     SetNumColumns   0    3
4     Rewind          0    8
5     Column          0    2
6     Callback        1    0
7     Next            0    5
8     Close           0    0
9     Halt            0    0
10    Transaction     0    0
11    VerifyCookie    0    14
12    Goto            0    1
13    Noop            0    0
```

sqlite3WhereBegin()
SELECT loop
sqlite3WhereEnd()

图9-10 一个不含WHERE子句的SELECT语句